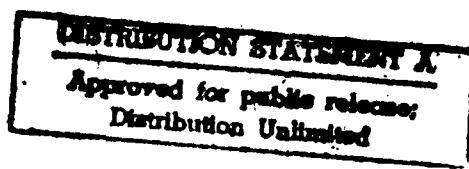
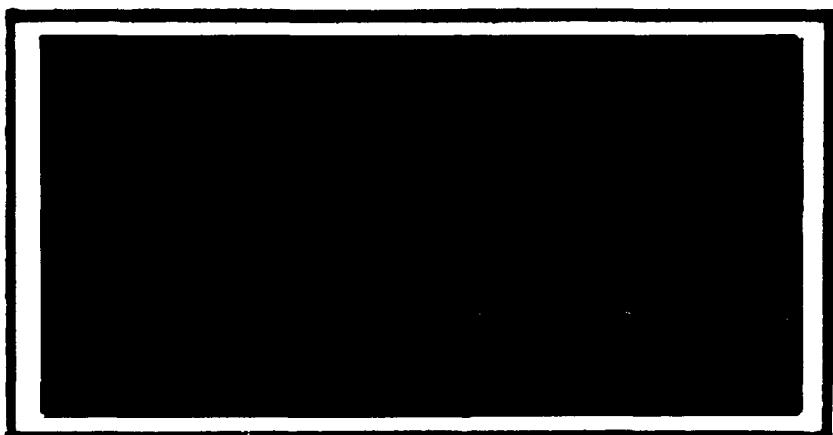


AD-A231 258

1



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

DTIC
ELECTED
JAN 22 1991
S E D

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/90D-07

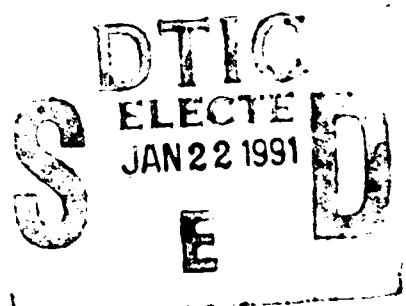
①

Graphical Representation of
Parallel Algorithmic
Processes

THESIS

Edward Michael Williams
Captain, USAF

AFIT/GCE/ENG/90D-07



Approved for public release; distribution unlimited

AFIT/GCE/ENG/90D-07

Graphical Representation of
Parallel Algorithmic
Processes

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Edward Michael Williams, B.S.

Captain, USAF

December, 1990

Accession For	
NTIS GRAAI	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution unlimited

Acknowledgments

The most important acknowledgement I must make is to the Lord Jesus Christ, who gives my life meaning, and without whom I would be forever lost.

My wife Jennifer and my sons Michael and Matthew deserve as much of the credit for this thesis as I do, since their sacrifices and loving support made this research possible.

Dr. Gary Lamont, my advisor, provided the challenge in this research by not simply accepting my results, but constantly asking "why?". He gave me enough rope to pursue different solutions, yet knew when to reign me in to prevent me from wasting too much time on dead ends.

The reason I advanced as far as I did with this research is due to the excellent system developed by Keith Fife for animating algorithms. I appreciate him listening to me during its development, which resulted in a system that was very well suited to this research effort.

The Sun workstations in the Graphics Lab, managed by Maj Phil Amburn, were very much appreciated; my opinion of any other computer system will be based on its comparison with the excellent environment he provided.

Finally, the rest of the graphics group: Dave Dahn, Phil Platt, Buck Stuart, and Bill DeRouchey. They kept me going when the times were rough, and made me go home when I'd been in the lab too long.

Edward Michael Williams

Table of Contents

	Page
Table of Contents	iii
List of Figures	vii
List of Tables	ix
Abstract	x
I. Introduction	1-1
1.1 Background	1-1
1.2 Problem	1-2
1.3 Objectives	1-2
1.3.1 Data Collection	1-2
1.3.2 Data Display	1-3
1.3.3 Algorithm Control	1-3
1.3.4 Algorithm Implementation	1-3
1.4 Assumptions	1-3
1.5 Scope	1-4
1.6 Summary of Current Knowledge	1-4
1.7 Approach	1-4
1.8 Summary	1-5
II. Requirements Analysis and Specification	2-1
2.1 Background	2-1
2.1.1 Program Visualization	2-1
2.1.2 Instrumentation	2-2
2.2 Requirements Analysis	2-2

	Page
2.3 Related Work	2-3
2.3.1 Seeplex	2-4
2.3.2 ParaGraph	2-4
2.3.3 Seecube	2-5
2.3.4 PICL	2-5
2.3.5 PRASE	2-5
2.3.6 Monit	2-5
2.3.7 AAARF	2-6
2.3.8 Other Work	2-6
2.4 Evaluation	2-
2.4.1 Criteria for Evaluation	2-7
2.4.2 Discussion	2-8
2.5 Specifications	2-11
2.6 Summary	2-12
III. System Design	3-1
3.1 Design Philosophy	3-1
3.2 System Description	3-1
3.2.1 Display System	3-3
3.2.2 Data Collection System	3-3
3.2.3 Algorithm Control System	3-3
3.3 System Integration	3-4
3.3.1 Display System	3-4
3.3.2 Data Collection System	3-7
3.3.3 Algorithm Control System	3-8
3.3.4 Communications	3-8
3.4 Summary	3-8

	Page
IV. Detailed Design and Implementation	4-1
4.1 Display System	4-1
4.1.1 AAARF Algorithm Class	4-1
4.1.2 Modifications to AAARF	4-20
4.2 Data Collection System	4-23
4.2.1 Instrumentation	4-23
4.2.2 Remote Collection Program	4-23
4.2.3 Modifications to PRASE	4-27
4.3 Algorithm Control System	4-29
4.4 Communications	4-29
4.4.1 Protocol	4-30
4.4.2 Connections	4-30
4.4.3 Communications Management	4-32
4.5 Summary	4-32
V. Parallel Algorithm Animation	5-1
5.1 Animation Process	5-1
5.2 Problem Domain	5-2
5.3 SCP Background	5-2
5.4 Animating the SCP	5-4
5.4.1 Analyzing the Algorithm	5-5
5.4.2 Displaying the Algorithm	5-8
5.4.3 Instrumenting the Program	5-10
5.4.4 Iteration	5-12
5.5 Ideas for Further Analysis	5-12
5.5.1 Advanced Methods	5-16
5.5.2 Additional Views	5-16
5.6 Scalability	5-16

	Page
5.7 Other Applications	5-17
5.8 Summary	5-18
VI. Conclusions and Recommendations	6-1
6.1 Evaluation of the Prototype	6-1
6.2 Conclusions	6-3
6.2.1 System Limitations	6-3
6.3 Recommendations for Future Research	6-5
6.3.1 Animation Views	6-5
6.3.2 AAARF Redesign	6-5
6.4 Summary	6-6
Appendix A. Parallel Shell Sort	A-1
Appendix B. PRASE Data Formats	B-1
Appendix C. AAARF Programmer's Manual	C-1
Appendix D. AAARF User's Manual	D-1
Appendix E. SCP Source	E-1
Appendix F. Source Code	F-1
Bibliography	BIB-1
Vita	VITA-1

List of Figures

Figure	Page
2.1. Typical Performance Analysis System	2-4
3.1. Relationships between the major components of the system	3-2
3.2. Object Structure of AAARF	3-5
3.3. Object Structure of an Algorithm Class	3-6
4.1. Objects implemented by the common library	4-2
4.2. Sample Master Control Panel	4-4
4.3. Performance Views Control Panel	4-7
4.4. Sample Status Panel	4-7
4.5. Original AAARF Child Process	4-8
4.6. Modified AAARF Child Process	4-9
4.7. Queue Data Structure for Algorithm Data	4-11
4.8. Sample Utilization View	4-14
4.9. Sample Gantt View	4-14
4.10. Sample Feynman View	4-15
4.11. Sample Communications Statistics View	4-15
4.12. Sample Communications Load View	4-17
4.13. Sample Queue Size View	4-17
4.14. Sample Message Lengths View	4-18
4.15. Sample Message Queues View	4-18
4.16. Sample Kiviat View	4-19
4.17. Sample Animation View	4-19
4.18. Algorithm Event Data Flow	4-21
4.19. Class Structure for Recording	4-22

Figure	Page
4.20. Typical instrumentation header file	4-24
4.21. Instrumenting the <code>main</code> node function	4-25
4.22. Instrumenting the host process	4-26
4.23. Connections between processes	4-31
5.1. Typical representation of a SCP search tree	5-9
5.2. SCP tree display for a 10x10 matrix	5-10
5.3. Event that uses no data (from <code>scpndcg.c</code>)	5-13
5.4. Event that uses existing data (from <code>scpndcg.c</code>)	5-14
5.5. Event with large data (from <code>scpcntcg.c</code>)	5-14
5.6. Event that requires derivation (from <code>scpndcg.c</code>)	5-15
B.1. PRASE trace data record	B-1
B.2. PRASE Structure Definition	B-2
B.3. Algorithm data record	B-5

List of Tables

Table	Page
2.1. Results of the Probe Effect Comparison	2-10
5.1. Sample 0-1 Matrix	5-3

Abstract

Algorithm animation is a visualization method used to enhance understanding of the functioning of an algorithm or program. Visualization is used for many purposes, including education, algorithm research, performance analysis, and program debugging. This research applies algorithm animation techniques to programs developed for parallel architectures, with specific emphasis on the Intel iPSC/2 hypercube.

While both *P-time* and *NP-time* algorithms can potentially benefit from using visualization techniques, the set of NP-complete problems provides fertile ground for developing parallel applications, since the combinatoric nature of the problems makes finding the optimum solution impractical.

The primary goals for this visualization system are: Data should be displayed as it is generated. The interface to the target program should be transparent, allowing the animation of existing programs. Flexibility – the system should be able to animate any algorithm.

The resulting system incorporates and extends two AFIT products: the AFIT Algorithm Animation Research Facility (AAARF) and the Parallel Resource Analysis Software Environment (PRASE). AAARF is an algorithm animation system developed primarily for sequential programs, but is easily adaptable for use with parallel programs. PRASE is an instrumentation package that extracts system performance data from programs on the Intel hypercubes. Since performance data is an essential part of analyzing any parallel program, views of the performance data are provided as an elementary part of the system. Custom software is designed to interface these systems and to display the program data.

The program chosen as the example for this study is a member of the NP-complete problem set; it is a parallel implementation of a general Set Covering Problem (SCP). {

Graphical Representation of Parallel Algorithmic Processes

I. Introduction

A significant problem with analyzing the behavior of algorithms executing on multi-processor architectures is that it is difficult to "see" what the data looks like or "watch" the activities of the processes in real time. Algorithm animation can provide a window so that the programmer or user can view the dynamic behavior of an algorithm and its data structures. Animation improves the understanding of algorithms and their application to specific problems by graphically representing the internal activities (both data and control flow) of an algorithm during its execution. Animation of parallel algorithms has the potential to help educators, students, and researchers attain a better understanding of complex algorithms and assist them in developing new ones.

1.1 Background

A program consists of algorithms that manipulate data structures. When designing a program, the programmer usually selects an algorithm, matches it with some appropriate data structures, and then tests it to determine whether the choices that were made are correct. This cycle repeats until the programmer is satisfied with the results.

The problem with this approach is in the test phase — the programmer has very limited information to evaluate the algorithm other than the results it produces. If the programmer could "look inside" the algorithm and data structures as the program is running, evaluating the algorithm's performance would be much easier. This is what algorithm animation does best. It provides the means to "look inside" the algorithm and actually *see* the actions as they are occurring.

With parallel processing systems becoming more prevalent, the problem of evaluating algorithms is much worse — determining what a program is doing is not just evaluating

the individual processes, but also their interaction — which can be nondeterministic. Understanding what is happening within an algorithm can be very difficult[23].

The human mind is a very specialized system — it is very efficient at interpreting and comprehending vast amounts of visual information. Algorithm animation can take advantage of this capacity by presenting the internal activities of an algorithm graphically and allow a greater understanding of the process than previously possible[26:3].

1.2 Problem

This is the problem — the programmer needs a way to “see” what is happening inside of a program in order to effectively develop, understand, and evaluate it.

Systems of this sort fall under the category of program visualization tools. One specific type of program visualization is called algorithm animation. Algorithm animation is the form of visualization that focuses primarily on displaying data and data structures, rather than control structures[12:19].

1.3 Objectives

The goal of this study is to develop an algorithm animation facility for parallel processes executing on different architectures, from multiprocessor systems to uniprocessor multitasking systems, with primary emphasis on the Intel hypercube.

To be useful, a parallel algorithm animation system must be able to:

- Animate any algorithm
- Allow users to interact with the algorithm
- Support the addition of new algorithms to the system
- Display animated data in real-time

The problem divides into the components described in the following paragraphs.

1.3.1 Data Collection Since the purpose is to observe the data structures within a program, a method must be developed for extracting appropriate data from within a

program as it is running. The objective is to get enough information from the program without significantly disturbing its normal functioning.

1.3.2 Data Display In order for the data to be easily interpreted it must be displayed in a neat, well-organized manner. Different ways of displaying the data are needed to provide different types of information from the data. The user should be able to select the desired displays and rearrange them on the screen as needed.

1.3.3 Algorithm Control As a minimum, the user should be able to reset, start, and stop an algorithm. In some cases the ability to control the execution speed of the algorithm under analysis could be useful. Another essential form of control is the ability to change the execution parameters of an algorithm.

1.3.4 Algorithm Implementation There must be an organized, structured interface between the animation system and the algorithm being implemented. The data collection routines must not interfere with the structure of the algorithm being implemented.

1.4 Assumptions

The following assumptions are based on conclusions drawn by Fife[12:4] and others:

1. Algorithm animations cannot be generated automatically. Since the operations performed by each type of algorithm are different or used in different combinations, automated detection and interpretation of these operations is not possible. A similar argument holds for developing the graphical views of the data. The same data can mean different things when presented in different ways. This does not rule out the use of an automated system for identifying obvious operations or events, but a programmer familiar with the algorithm is needed to complete the task.
2. Any algorithm can be animated. All algorithms manipulate data structures, so there is always some possibility for displaying that data in order to gain insight into the operation of the algorithm.

1.5 Scope

The object of this study is to develop techniques for visualizing the operation of programs. Different algorithms are implemented to test these techniques, but algorithm research is not part of this study. The result of this research is a prototype system for animating parallel algorithms.

1.6 Summary of Current Knowledge

Only recently has any work in parallel algorithm animation been reported. All of the work has been in animating specific algorithms or specific types of data. Displaying parallel system performance data is where most effort seems to be directed. Other areas of activity include animating parallel matrix manipulations. Since each of these systems could be considered to be a special case of a general parallel algorithm animation system, they are good candidates for a starting point in developing a general system.

One system, Seeplex from Tufts University[8], is currently capable of real-time display of data from a parallel system. Unfortunately, its use is limited to the NCUBE hypercube, a system that is not available at AFIT. The rest of the systems perform off-line data analysis. All of the systems contain two common components: data collection routines that obtain data from the processes running on the parallel system and a display program that runs on a graphic workstation.

There are several systems that can be considered a special case of parallel algorithm animation. These systems provide a solid foundation for developing a general system for animating parallel algorithms; they are discussed in more detail in Section 2.3.

1.7 Approach

Recognizing the significant effort already invested in the area of program visualization and parallel processor performance analysis, the design of the prototype animation system emphasizes the use of existing systems as a foundation and developing new software to add new capabilities.

Several existing systems were evaluated and the most suitable one chosen for the foundation. The systems were evaluated using these criteria:

Availability: In order for the system to be used, it must be available in source code form and have no licensing requirements or restrictions on its use. The necessity of available source code is based on the assumption that the system will require modification to form a part of the algorithm animation system. There can be no restrictions or licensing requirements since the resulting animation system will be in the public domain.

Environment: The system must be able to run in an environment that is available at AFIT (this includes computer type, operating system, and windowing environment).

Expandability: The design of the system must be capable of being extended to meet the new requirements. This also includes documentation — it is difficult to extend a design if there is no explanation on exactly what the original design is.

Once the foundation system is chosen, techniques from other systems are used to fill in as many gaps as possible. This provides a base for the parallel algorithm animation system. The remaining parts of the system are developed using a high level language and software engineering principles.

The C programming language[21] is the most likely language since it is the primary language supported by the UNIX operating system[32, 2] available on the computer systems at AFIT.

The system is tested by implementing several algorithms, concentrating on parallel search programs. This class of programs is interesting due to its exponential growth in search time as the size of the problem increases. These problems are good candidates for improved performance using parallel techniques.

1.8 Summary

The goal of this thesis investigation is to develop the means to "see" what is occurring during the execution of a program on a parallel system. Emphasis is placed on the Intel

hypercube, due to its availability.

The next chapter contains the requirements definition for the animation system, a brief review of program visualization and parallel program instrumentation, the literature review, and an evaluation of existing systems for use in this research. Chapter III describes the design of the prototype animation system. The implementation of the prototype system is presented in Chapter IV, and the process of animating an algorithm is described in Chapter V. Conclusions and recommendations are presented in Chapter VI.

Included in Appendix A is the source code for a parallel sort program discussed in Chapter II. The data formats related to PRASE are contained in Appendix B. Appendix C contains an updated programmer's manual for AAARF, and Appendix D contains an updated user's manual for AAARF.

II. Requirements Analysis and Specification

This chapter describes the requirements analysis process. First, a brief review of program visualization and instrumentation is provided for background. The next section describes the characteristics of a real-time parallel algorithm animation system, the following section reviews current work in the area of parallel program visualization, and the final section provides an evaluation of the currently available systems for use as a foundation for this research.

For the sake of consistency and standardization, the formal definition of an algorithm, the terms and definitions relating to algorithm animation, and the algorithm animation system model described in Fife[12:8-17] is used throughout this chapter and the rest of this thesis.

2.1 Background

The purpose of this section is to introduce terms and definitions for the two subject areas that form the basis for this research: program visualization and instrumentation of software. If additional information is needed, the references provide a more detailed discussion of these subjects.

2.1.1 Program Visualization Program visualization is the use of graphics to illustrate some aspect of a program's execution[26:2]. This is a very broad definition, but within program visualization there are four major categories based on the type of information displayed and the method used for display:[26:15]

- Static display of program code
- Static display of data
- Dynamic display of program code
- Dynamic display of data

The last category is also called *algorithm animation*[12:19].

There is much research being done in program visualization, and Fife[12] and Meyers[26] review many of the systems available now.

2.1.2 Instrumentation The term *instrumentation* is usually used in reference to electronic equipment that is used to monitor the conditions within a circuit or system. When applied to software, the term means very much the same. Instrumentation in software consists of modules that are inserted into the user's code to monitor an activity and send data out to another system so that it can be displayed. An undesirable side effect of instrumentation is that it can disrupt the timing and performance of the program it is monitoring. An in-depth discussion of the effects of instrumenting software is presented by Marinescu[25].

2.2 Requirements Analysis

In addition to the requirements described by Fife for an algorithm animation system for sequential algorithms[12:23-26], the parallel algorithm animation system should have the following requirements:

1. The ability to animate programs running on another host. This is necessary since the parallel system may not have the ability to support the graphics requirements of the animation system. This includes the ability to start and stop the algorithm on the remote system.
2. A built-in capability to display performance data from the parallel system. System performance is an essential part of analyzing any parallel algorithm, so it should be part of the basic animation system.
3. The animation system must not place limitations on the structure, size, or configuration of the program running on the parallel system. This requirement is based on the second assumption in Section 1.4 — if any program can be animated, then the system should be able to animate it.
4. The animation system should display data from the algorithm in real time. Real time doesn't mean that the display needs to be updating as fast as the data is

being generated on the remote host; that may not be very useful, or even practical. This requirement only specifies that the displaying of the data begin before the target program has ended. This allows the user to get immediate feedback from the parallel program; program parameters can be adjusted and anomalies detected without having to wait for the entire run to complete. This can amount to large time savings, especially when each program run takes a long time.

5. The system must be able to animate existing programs. Since the animation system can be used for research and education, it is advantageous to be able to take programs that were developed independent from the animation system and animate them so that their behavior can be observed and evaluated. This requirement is also beneficial when using the animation system as a debugging tool — the final program will not be a part of the animation system, so it shouldn't have to be designed around the animation system.

2.3 Related Work

Until very recently, there has been little work in animating parallel algorithms. In April 1990 at the Fifth Distributed Memory Computing Conference, a paper was presented by Couch[8] that described Seeplex, a system for real-time performance analysis of hypercube programs. In addition, three systems that perform offline analysis were presented[16, 27, 30].

Most of the work that is related to parallel algorithm animation is in the area of performance analysis of parallel computing systems. These systems are usually divided into a data collection monitor that is incorporated into the user's programs, and an offline report generator or viewing program. Figure 2.1 shows a typical arrangement for this type of system. This process can also be thought of as an animation system that is displaying the activities of the operating system instead of an application program.

The following sections describe the systems mentioned above, and others that are related.

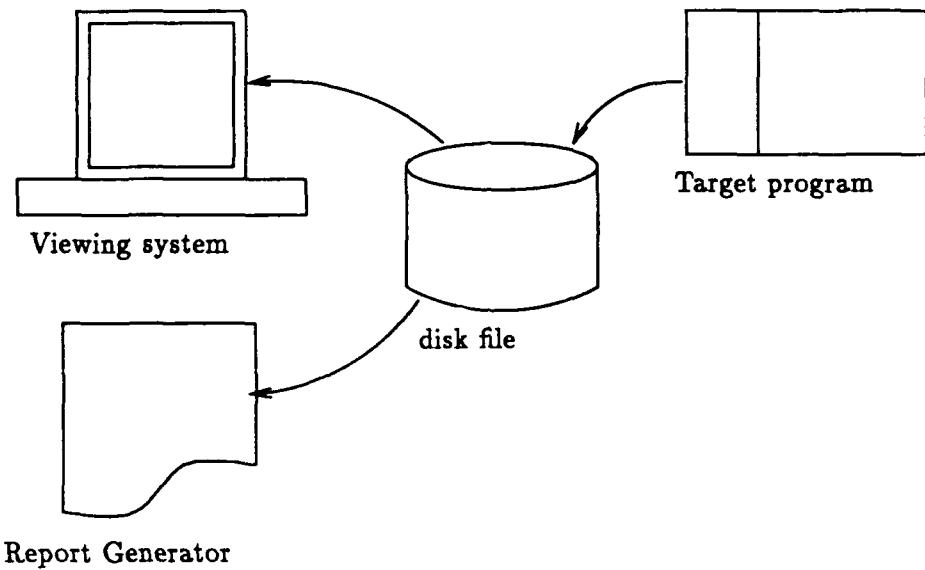


Figure 2.1. Typical Performance Analysis System

2.3.1 Seeplex Seeplex[8] is a real-time performance monitor for hypercube systems. It was developed from Seecube (described below), but has a major difference — instead of getting the performance data from monitors inserted into the user's code, the data is extracted directly from the custom operating system on the nodes of an NCUBE hypercube. The available displays include node activity, communication channel activity, and node computation rates.

The display software is implemented on a Sun workstation and is capable of controlling the program execution, the type and quantity of monitoring data produced, and the views of the data that are used.

2.3.2 ParaGraph ParaGraph is a system that was developed by Mike Heath at the Oak Ridge National Laboratory (ORNL) Mathematical Sciences Division[16]. ParaGraph is not the product of a distinct research effort; it is the result of an ongoing effort to develop a tool to aid other research. It accepts performance data from PICL (described below) and displays it in many ways. It provides node activity and node CPU utilization, as well as high-level and detailed views of message passing activity between nodes. ParaGraph

operates in a post-processing mode, providing an offline analysis capability.

ParaGraph is implemented under the X Window system, giving it the capability to run on many different host systems.

2.3.3 Seecube Seecube is a system designed at Tufts University for monitoring activity within a hypercube architecture[7]. The system is divided into three pieces - the data collector, the resolver, and the graphical display driver. The data collection routines are inserted into the programs that run on the nodes of the hypercube. They collect event information and send it to the host where it is stored for later analysis. The resolver takes the collected data and correlates it so that corresponding reads and writes are matched up and everything is in time order. The display driver provides the means of viewing the data in many different formats.

The data collection routines are implemented for the NCUBE hypercube and the resolver and display components run on a Sun workstation.

2.3.4 PICL The Portable Instrumented Communication Library (PICL)[15] is another tool developed at ORNL. PICL is a dual-purpose system; its original function was to provide a portable, consistent communications interface for distributed-memory multiprocessor systems. PICL was later expanded to provide data for performance evaluations, algorithm analysis, and algorithm animation. The data produced by PICL is used as input to ParaGraph, an off-line animation package also developed at ORNL. ParaGraph provides up to ten different views of the data, and allows multiple views to be active simultaneously.

A complete implementation of PICL is available for several systems, including the Intel iPSC/2, the Intel i860 hypercube, and the NCUBE. An implementation for the Intel iPSC/1 hypercube is under development. The ParaGraph animation system is implemented under the X Window system.

2.3.5 PRASE This system was developed at AFIT by Capt Mark Kahl[19]. It is based on the Seecube system. It replaces the Seecube data collection routines with new ones that are smaller, faster, and produce data that is more compact. PRASE provides a translator so that the Seecube display driver can be used to view the data. A unique

feature of PRASE is that the user is not burdened with inserting calls to the data collection routines — preprocessors are provided that find the calls to the system functions that the user wants to monitor and substitutes calls to the data collection routines.

PRASE has been implemented on both the Intel iPSC/1 and iPSC/2 hypercubes.

2.3.6 Monit While the previously described systems monitor operating system level activities, Monit is designed to monitor at a slightly higher level[22:163-174]. It was developed by Microelectronics and Computer Technology Corporation, Austin, Texas, to monitor the activities within the run-time kernel of a custom programming language [22:165]. It also provides a data collection portion and a post-processor to display the data.

The target machine for this programming language is the Sequent 8000 multiprocessor system, and the display program is implemented on a Sun workstation.

2.3.7 AAARF The AFIT Algorithm Animation Research Facility (AAARF) was developed by Capt Keith Fife[12]. In its current state, it is used for animating sequential algorithms; however, it was designed so that it could easily be extended for use with parallel algorithms.

AAARF is implemented on a Sun workstation using the SunView windowing interface. It uses the event-driven nature of the SunView system to manage multiple algorithms simultaneously, as well as multiple views of a single algorithm. The displayed data is produced by an instrumented program that runs as a child process under AAARF. It is this feature of AAARF that makes it suitable for adapting to parallel algorithm animation: there is no limitation on how the data is generated — it could be from a process on the local system, or a process on another system feeding data through a communications link to AAARF.

2.3.8 Other Work Others have developed similar types of animation systems, including Rover and Wright[30], and Paul and Poplawski[27]. These systems take data from an instrumented program, filter, sort, or otherwise preprocess it, then use another program to interpret the data visually.

Another area of significant work is in animating sequential algorithms. Many systems exist that can animate a sequential algorithm, including BALSA[6] and TANGO[31]. Fife provides a summary of this class of animation system in his research[12:18-22].

2.4 Evaluation

The systems described in the previous section all perform a function similar to real-time parallel algorithm animation, yet each one differs slightly. This section evaluates the available packages for use as a starting point for this research.

2.4.1 Criteria for Evaluation Since none of the packages described above completely meet the requirements in Section 2.2, it is assumed that any package selected for use needs to be at least modified and possibly extended. The criteria used for evaluating the packages must account for this, but also needs to take into account how well the package already meets the requirements.

Certain requirements must be met for a package to even be considered:

1. Source code must be available
2. There can be no licensing requirements or restrictions for use
3. The program must be usable in an environment that is available at AFIT

Specific criteria for evaluating all packages include:

1. The package must be well-documented, including user's manual and source code documentation
2. The design of the system must be capable of being extended

These items are subjective and indicate the amount of effort that would be required if the package is selected for use. The candidates for the data collection system are also evaluated on these factors:

1. Probe Effect — how much impact do the collection routines have on performance of the program under observation

2. Intrusiveness — what effect do the collection routines have on the structure of the program being monitored
3. Source Code Complexity — how easy is it to make any necessary modifications

The first two factors determine the impact that the collection routines have on the program under test. In order to get valid results, the impact should be as small as possible. Once again, the last factor serves as an indicator of how much effort would be required to use this package.

2.4.2 Discussion Applying the first set of criteria eliminates some packages due to hardware and software incompatibilities. Unfortunately, one of these packages is Seeplex, the only package that currently has a real-time capability. Seeplex is designed to get its trace data directly from a custom operating system on the NCUBE hypercube, which is not available at AFIT. Monit is also eliminated because its source of data is not available at AFIT.

The remaining packages do not contain both the data collection and display components, so the evaluation of display packages and data collection packages is discussed separately.

2.4.2.1 Display Component With one exception, the remaining display packages are designed to display performance data. Seecube and ParaGraph both present different views of the performance data and have no capacity for displaying algorithm-specific data without extensive modifications throughout the program to interpret and display the new type(s) of data.

AAARF is the exception — it was designed to display algorithm specific data. Unfortunately, there is no built-in provision to display performance data, but performance displays can be developed in the same way an algorithm specific display is created.

The quality of the displays produced by the three candidates is approximately equal, although the format and content of the Seecube displays is more difficult to interpret than the ParaGraph displays.

Regardless of the choice of packages, displays for algorithm specific data need to be created. With AAARF, the performance displays also need to be created, since AAARF has no predefined displays. This, however, is one of AAARF's strengths — flexibility. It allows the programmer/user to develop the displays exactly as they need to be for each program.

2.4.2.2 Data Collection Component There are only two data collection packages that meet the conditions for candidacy: PRASE and PICL. The rest require hardware that is not available at AFIT.

In a sense, comparing these two packages on an equal basis is not fair to either program because they were developed for different purposes: PICL's primary function is as a portable communications model that can be implemented on many different architectures without modifying anything except low-level communications primitives; PRASE was developed to instrument programs for one specific system and one communications model. Both packages, however, produce similar types of trace data from programs running on an Intel iPSC/2 hypercube.

To measure the probe effect when using the two packages, a simple parallel shell sort was instrumented. The execution of the program was timed; the results are in Table 2.1. Times are in milliseconds. To avoid any delays that may be caused by the host activity, the time is measured from just after the node program receives its part of the data from the host until just before it sends its part of the data back to the host. There is very little difference between the times from the original uninstrumented program and the two instrumented ones.

The overhead due to the instrumentation is based on the number of messages sent between nodes. The sorting algorithm used sends the same number of messages regardless of the size of the data — that is why the overhead seems to be independent of the size of the data. The source code for the shell sort is included in Appendix A.

Ideally, there should be no change to the program under test that is a direct result of instrumenting it. Once again, the two packages take different approaches on this topic. PICL presumes that the programmer uses its library when developing applications, then

Table 2.1. Results of the Probe Effect Comparison

Elements Sorted	Original Sort*	Instrumented With PRASE*	PICL*
1024	50	58	58
2048	85	101	95
4096	167	179	176
8192	376	386	392
16384	726	735	732
32768	1565	1577	1573
65536	3321	3329	3327

*Time in milliseconds

turns on the data collection when needed. The intrusiveness of this is small because the program is designed around the data collection package. On the other hand, if the library doesn't directly support the methods the programmer wishes to use, the programmer has to change methods or find another way to approach the design of the program. This is quite intrusive.

PRASE is very transparent to the program being tested. It has a preprocessor that takes the program source code and inserts its own initialization code and calls to the collection routines. The user only has to put a special comment in the source to let the preprocessor know where to put the initialization code. This lets the programmer develop the program and later instrument it to analyze its activities; and when the analysis is done, the program is not burdened with the added overhead of the inactive instrumentation.

The PICL source is quite complex due to its primary purpose: portability. It is constructed in a layered manner so that only the bottom layer needs to be replaced when porting the package to a new system. This generic approach makes the code more complex because a given operation can require many different functions in different places. PRASE is targeted to one architecture — the Intel hypercube. This allows the collection routines to be simple and concise.

The documentation for the two packages also varies widely. PICL has excellent detailed documentation; the reference manual is well organized, and the source is appropriately commented. The reference manual is not complete — the high-level functions are not documented, but these are not required to do most communications tasks. By way of

contrast, the documentation for PRASE is limited in nature and not up to date. There is a User's Manual [20], but it wasn't updated when PRASE was ported to the iPSC/2 from the iPSC/1. There is no technical manual other than Mark Kahl's thesis [19], which is at too high of a level to be useful for modifying or enhancing PRASE. The data collection routines in the file `prase_code.c` are mostly unstructured, and the source code comments are directed at the level of a non-programmer, not at the level of a programmer familiar with hypercube programming.

2.5 *Specifications*

The display package that best fits the requirements is AAARF. The advantages of using AAARF are:

- It is currently capable of real-time algorithm animation
- It was developed at AFIT and source is available
- It was designed to be flexible to encourage modifications and extensions
- The documentation is excellent, both manuals and source code

The only major disadvantage to using AAARF for the display component is that it has no pre-defined library of views. This is somewhat offset by the well-defined interface AAARF provides to simplify the task of creating views. The performance views from ParaGraph are very well done, and can serve as patterns for some of the views for AAARF.

In spite of its poor documentation, the selection for the data collection component is PRASE. Since the performance impact of the two packages is nearly identical, the choice is based on the subjective factors of *intrusiveness* and *ease of use*. PRASE's transparent nature makes it simple to apply to existing programs, while PICL requires redesigning the program around its library. The simple nature of PRASE's source code overcomes its poor documentation and makes it easy to modify.

2.6 Summary

This chapter has detailed the requirements for the real-time parallel algorithm animation system. Current work was reviewed and evaluated for use as a foundation for this research; there were no systems that completely met the requirements, but a combination of AAARF and PRASE provides enough existing functionality and expandability that together they provide a good place to start. The integration of these packages into the algorithm animation prototype is discussed in the next chapter.

III. System Design

This chapter presents a high level design for the real-time parallel algorithm animation system prototype. The detailed design is covered in Chapter IV.

3.1 Design Philosophy

The philosophy supporting the design of this system is to use existing software as much as possible. The primary reason is to avoid duplication of effort. There is no need to spend much time and effort creating a new tool if it has already been done by someone else. The existing works may need to be modified, but the effort is minimal compared to recreation (and documentation).

The object of this design is to assemble a system that meets all the requirements using existing packages where possible and developing new software as needed.

3.2 System Description

As was discussed in Section 1.3, the design of the parallel algorithm animation system decomposes into four major components:

1. Data Collection
2. Data Display
3. Algorithm Control
4. Algorithm Implementation

The first three components comprise the animation system itself, while the last involves the use of the system to animate an algorithm; the discussion of this topic is contained in Chapter V.

The requirements in Section 2.2 to a great extent determine the structure of the animation system. The first requirement states that the animation system must be capable of animating a program running on another host; this splits the system into at least two

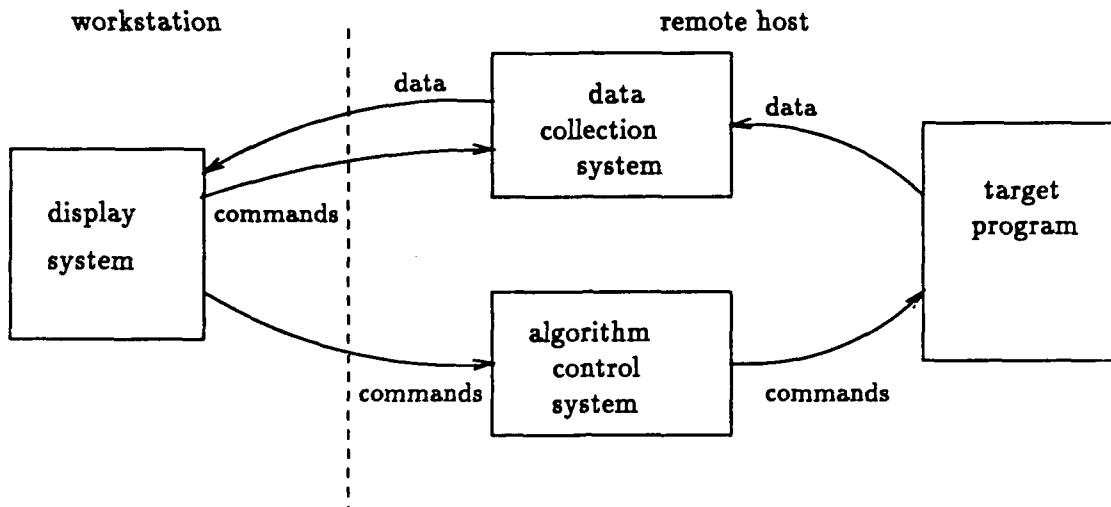


Figure 3.1. Relationships between the major components of the system

parts, one running on the workstation and one on the remote host. The fourth requirement requires real-time display of data from the algorithm — this requires that the parts be in continual communication with each other, and precludes the post-processing mode of operation.

The third and fifth requirements simultaneously make the system more difficult to design but make it significantly more useful — these requirements specify that there is to be minimal interference to the program being animated. This gives the system the capability to animate an existing program — as is done in Chapter V. The difficulties arise from the fact that the animation system can assume nothing about the structure of the program, the methods used to acquire the hypercube system, or the method used to load the nodes with their programs. To properly handle as many different possibilities as possible, the portion of the system on the remote host is divided into two major parts: one to collect the data from the program and send it to be displayed, and the other to control the program itself.

The design of each of the major parts of the animation system is discussed in the following sections. Figure 3.1 shows the major components of the system and the interaction among them.

3.2.1 Display System This component of the animation system is responsible for displaying the data being collected from the program on the remote system. The second requirement in Section 2.2 specifies that displays must be provided for performance data, since performance analysis is an essential part of analyzing any parallel program. Additional displays may also be required to display data specific to an algorithm or a group of algorithms.

The *display system* also provides the user with the capability to interact with the algorithm, including setting program parameters, starting the program, and stopping it. This component of the animation system runs on the graphic workstation used to display the data.

3.2.2 Data Collection System This is the component that contains the instrumentation that is inserted into the program being animated. In order to meet the third requirement, the instrumentation should be transparent to the target program — few changes should be necessary to the target program to insert the instrumentation.

In the current implementation of the operating system on the nodes of an Intel hypercube, the nodes cannot directly access systems outside the hypercube. This requires a separate process running on the hypercube host processor that collects the data generated by the instrumentation and forwards it to the *display system*. Even if the nodes could communicate directly with other systems, it may not be advisable to burden the node processors with the overhead of network communications, making the separate process useful for performance reasons. This is the only component of the system that is required to run on the remote system.

3.2.3 Algorithm Control System All requests to start and stop the target program are executed by this component. This component must be independent of any algorithm, but must be flexible enough to control most, if not all, programs. This component can reside on any system, but should run on the remote system to allow easier control of the target program. To meet the fifth requirement in Section 2.2, the *algorithm control system* should not eliminate the ability to start the program on the remote host manually —

programs may have an interactive user interface. This does, however, remove the capability to stop the program from the *display system*.

3.3 System Integration

As was stated in Section 1.7, the design of this system will incorporate existing works as much as possible. Section 2.4 evaluated the systems described earlier in that chapter as candidates for use in this research. The systems that were decided on in Section 2.5 are the AFIT Algorithm Animation Research Facility (AAARF)[12] for the *display system* and PRASE[19] for the *data collection system*. The rest of this section discusses the modifications and additions necessary to integrate these systems into the parallel algorithm animation system.

3.3.1 Display System AAARF already has the capability to support the requirements described above for the *display system*. The parallel animation system can be implemented as an algorithm class[12:14] under AAARF's management.

AAARF was designed using Object-Oriented Design (OOD) principles. There are four basic classes that were used to build AAARF[12:40-45]:

Windows: This class represents a rectangular area on the display that can contain text, graphics, and other windows. Windows can be moved and resized.

Panels: This class is a specialized subclass of windows. Panels provide a means for the user to monitor and modify system parameters. Panels cannot be resized.

Menus: The menu class provides the user an opportunity to select from a list of options.

Components: A component is one piece of the interface to an implementation of an algorithm. The three special cases of component are *input*, *algorithm*, and *view*.

Figure 3.2 shows a high-level view of the relationships between the objects for AAARF. An algorithm class is actually an instantiation of an algorithm window. The object relationships for an algorithm class are shown in Figure 3.3.

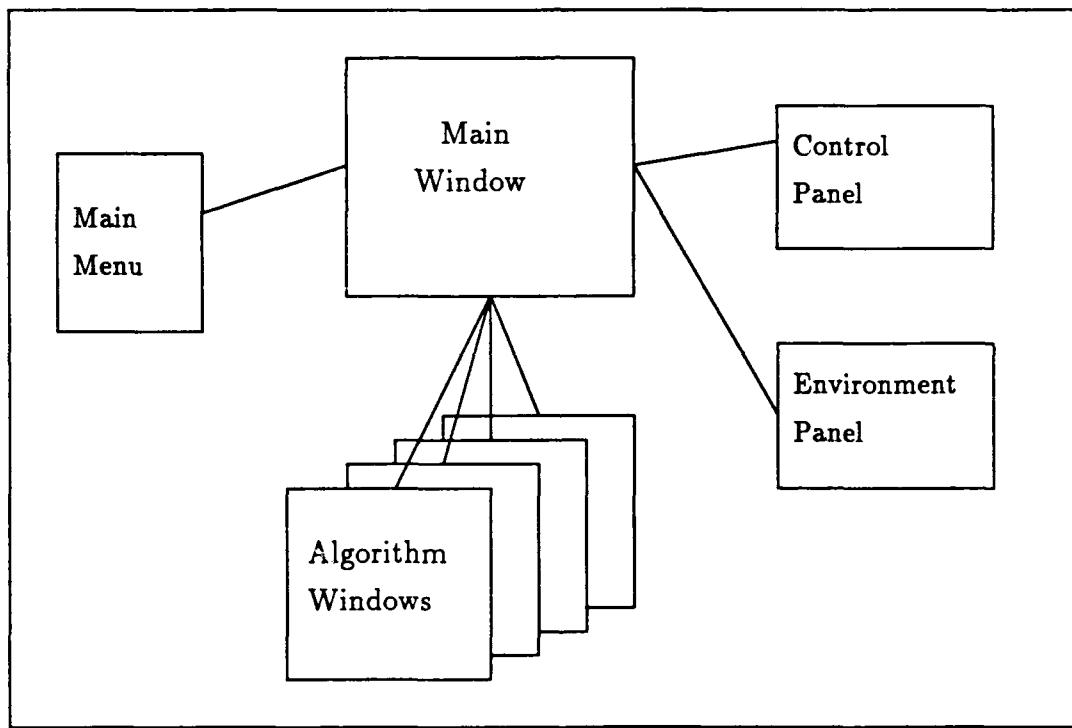


Figure 3.2. Object Structure of AAARF[12:45]

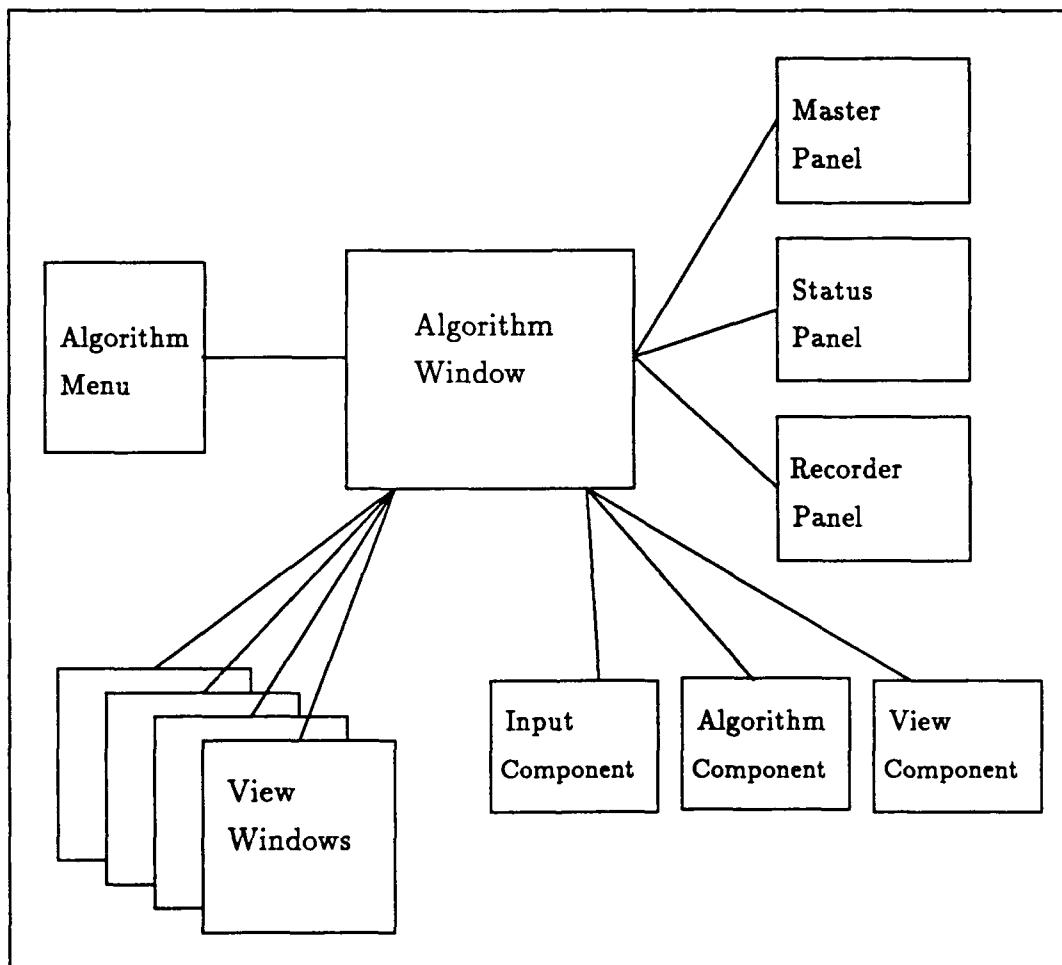


Figure 3.3. Object Structure of an Algorithm Class[12:46]

Since a large part of creating an algorithm class is the same, no matter what the algorithm, AAARF provides a *common library*[10:26] that implements most of the common parts of each object, leaving the programmer to fill in the details for the specific algorithm being animated. This approach is advantageous because the common library mostly shields the programmer from having to deal with SunView (the windowing system) except for the actual drawing commands used to create the views. The requirements for implementing an algorithm class are described in *The AAARF Programmers Manual*[10:41].

3.3.2 Data Collection System The PRASE system forms the core of the *data collection system*. In its current form, PRASE consists of the instrumentation routines that are inserted into the target program, and a separate program that runs on the hypercube host processor that collects the data from the instrumentation routines and writes it to disk files (one for each node) for later processing.

The instrumentation routines themselves intercept system calls, write essential information to a trace buffer, and then call the system routines. The PRASE routines are quite efficient and complete — few modifications are necessary.

To support the remote hosting of the algorithm as well as the real-time requirement, the separate program that collects data from the instrumentation routines needs modification — the data needs to be sent directly to the *display system* instead of a disk file. At this point, any unneeded data can be filtered out to reduce the communications load.

The existing collection program is not concerned with the time information contained in the instrumentation data, since all time-ordering is done at a later stage in the processing for that system. The real-time requirement for this animation system makes it necessary to time-order the data so that the *display system* gets the data in the order it is generated. The buffering that occurs in the communications system on the hypercube can cause data arriving from the node processors to arrive out of the proper time sequence; some method must be implemented to ensure that the data is sent to the *display system* in proper time order.

3.3.3 Algorithm Control System There is no existing product or system that meets the needs of this component of the system. It needs to be designed to interface directly with the *display system*, as well as the operating system so that it can control the target program.

3.3.4 Communications While not an actual component of the animation system, the communications system plays an important part in the design. The system is divided into several independent pieces that must communicate with each other; there are three different communications environments that are encountered:

1. Within the hypercube
2. Between processes on the same system
3. Between processes on different systems

The communications environment inside the hypercube consists of 10MB Ethernet links between nodes, and a multiplexed Ethernet connection between the host and all nodes. The communications between the nodes is quite fast due to the direct links and specialized communications hardware on the Intel iPSC/2. The links between the nodes and the host, however, are relatively slow.

Communications between processes, whether local or remote, depend heavily on the operating environment. Fortunately, UNIX provides a standard set of communications primitives that allow a consistent communications environment across multiple systems.

In the case that the instrumentation produces data at a higher rate than the communications system can deliver it, there must be a mechanism to prevent the resulting data backups from significantly affecting the operation of the target program.

3.4 Summary

This chapter describes the packages that form the foundation of the parallel algorithm animation system, the changes that need to be made to them, and the additional software that needs to be designed to integrate them into the animation system. The packages were

chosen so that few changes are required to interface with each other. The new modules that need to be developed are an algorithm class for AAARF, a program to collect the data on the remote system and send it to the *display system*, and a program to control the target program. The next chapter details the modifications required to AAARF and PRASE and the design of the new modules.

IV. Detailed Design and Implementation

This chapter describes in detail the design and implementation of the parallel algorithm animation system. This covers the changes to AAARF and PRASE that are required as well as the new programs and modules that need to be developed.

The three main components of the animation system are discussed separately. The discussion of the display component covers the development of the algorithm class for AAARF and the changes to the existing AAARF software. Section 4.2 describes the changes that are necessary to the PRASE software and Section 4.3 discusses the design of the algorithm control process that runs on the remote system. The communications system is an integral part of all the components, but it is described in a separate section to reduce repetition and permit a more comprehensive discussion of the methods used.

4.1 Display System

4.1.1 AAARF Algorithm Class As described in Section 3.3.1, an algorithm class under AAARF is an instantiation of an algorithm window. The composition of an algorithm window is shown in Figure 3.3. The AAARF common library provides most of the functions required to implement the algorithm class — Figure 4.1 shows which objects are implemented in the *common library*, and the rest are partially implemented. To complete the implementation of the algorithm class, only those functions that specifically deal with the algorithm being implemented need to be developed. This includes not only the functions that put the displays on the screen, but also ones that process the data being received from the algorithm, and others that maintain any data structures internal to the algorithm class. The common library was designed such that the programmer designing and implementing the animation can concentrate primarily on developing the displays instead of being distracted by the details of the environment.

Here are the objects left incomplete by the common library[10:47-50]:

- Master Panel

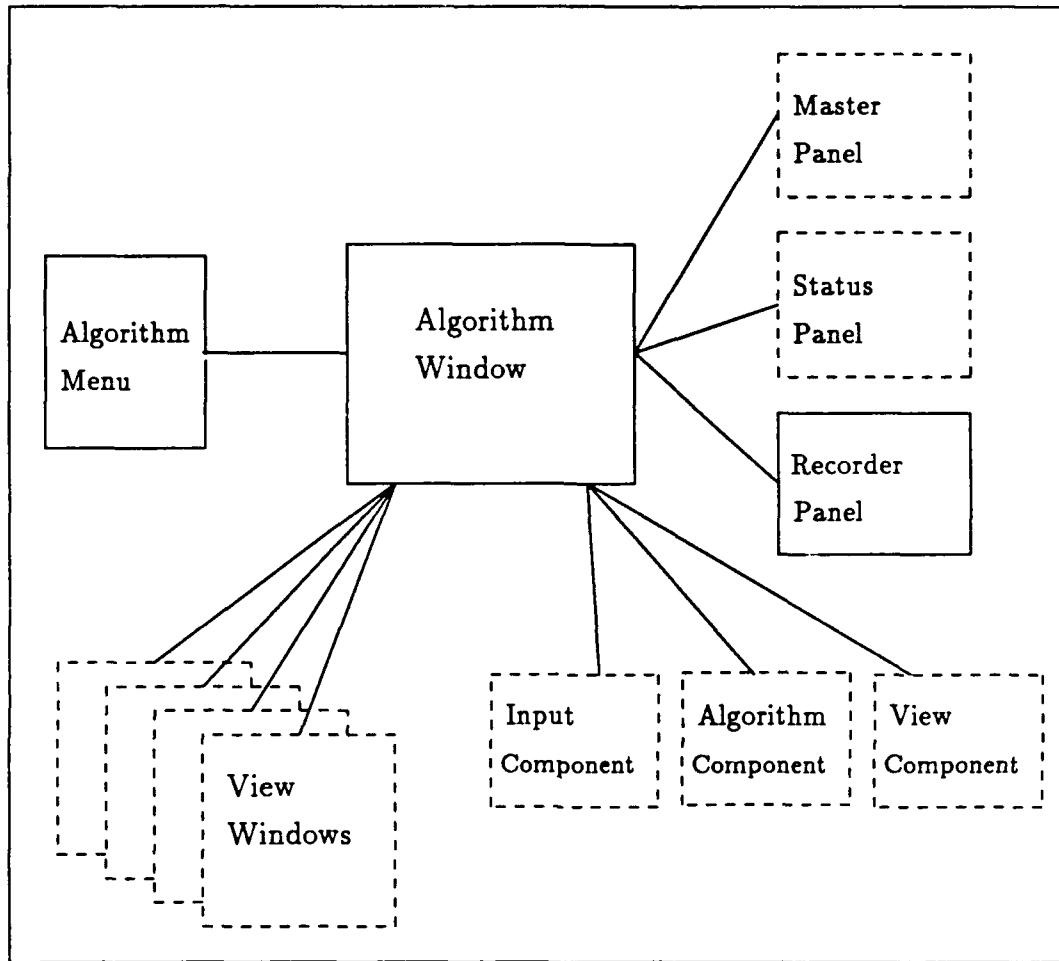


Figure 4.1. Objects implemented by the common library

- Status Panel
- Algorithm Component
- View Component

In addition to completing the objects listed above, the common library needs some global variables defined[10:45]:

- The name of the algorithm class
- An icon to use for the class

- Parameters for the Speed control
- Size of one event packet from the algorithm

Some of the functions and data structures required to complete the algorithm class are the same, no matter what parallel algorithm is being animated. This is because the *data collection system* is always used to provide the event data; the raw performance data is always in the same format. This makes it possible to create a library of common functions that can be used for all parallel animations. Since the library is mostly concerned with collecting and displaying performance data, the library is called the *parallel views library*. The library contains the entire algorithm component and parts of the view component. This simplifies the job of animating a parallel algorithm by allowing the programmer to focus on developing the algorithm data views without being concerned about the details of getting the data from the algorithm and displaying performance data. The *parallel views library* is fully documented in the updated *AAARF Programmer's Manual* contained in Appendix C.

The rest of this section is devoted to describing the development of the objects required to complete the algorithm class. The development of the components contained in the *parallel views library* is discussed in the appropriate section below.

4.1.1.1 Master Panel The Master Control Panel (MCP) is the central point for controlling all aspects of the algorithm class' operation. A sample MCP is shown in Figure 4.2. Some of the items on the panel are common to all animations, and these are set up by the common library. The input parameters can be different for each algorithm, so they must be provided for each algorithm. The selection of which algorithm within a class also must be provided. The breakpoints to allow and the list of available views also cannot be put into the common library. Since items are added to this panel, help for these items also is needed.

The specific functions required to complete this object are:

- `addInputSection()` — adds the items to the MCP that accept the input parameters for the algorithm

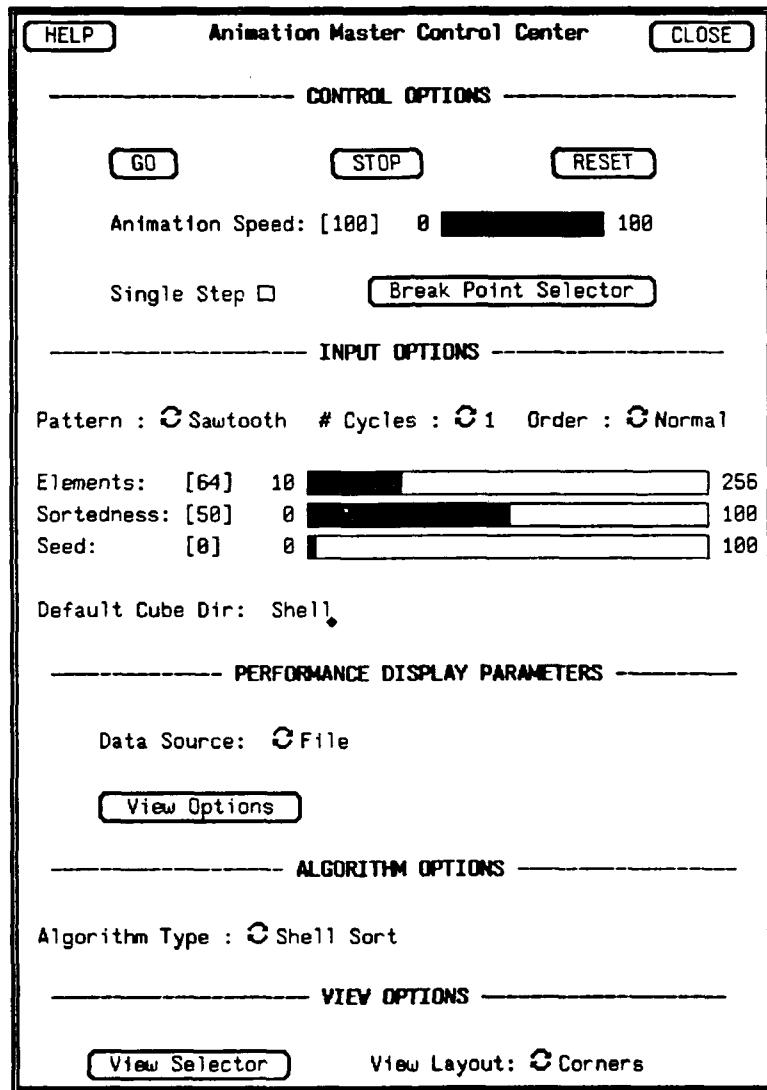


Figure 4.2. Sample Master Control Panel

- `getInputParameters()` — reads the input panel items to get their current values
- `setInputParameters()` — sets the input panel items to saved values
- `addAlgorithmSection()` — adds the panel items that select the algorithm within the class
- `getAlgorithmName()` — provides a string containing the name of the current algorithm
- `getAlgorithmParameters()` — reads the algorithm panel items to get their current values
- `setAlgorithmParameters()` — sets the algorithm panel items to saved values
- `setBreakPointItem()` — adds choices to the pre-existing breakpoint panel item
- `setViewItem()` — adds the available views to the pre-existing views panel item
- `masterHelp()` — fill a help panel with text describing the items added to the panel

The MCP shown in Figure 4.2 is from an animation of a simple parallel sorting algorithm. The top section controls the execution of the animation and is provided by the common library. The input section shown here contains items that control the random array generator used in the sort program. It can also contain parameters that select the data set to work on, the number of processors to use, and other types of data that need to be provided to the program under observation. The next section on the panel contains the options for the *performance view library*. This section is provided as a basic part of the parallel animation system by the performance view library. The button labeled “View Options” activates another panel that contains all the controls for the performance views. This panel is shown in Figure 4.3. The bottom section controls the layout of the animation windows. The panel items are created by the common library, but the list of available views comes from the performance view library and the algorithm specific views.

4.1.1.2 Status Panel The status panel's purpose is to provide a place for the animation to display textual information about the status of the algorithm. The status panel from the parallel sort animation is shown in Figure 4.4. The only predefined part of

this panel is the panel itself — since the contents of the panel are algorithm-dependent, the programmer must create and maintain the panel.

These are the functions required to complete this object:

- `buildStatusDisplay()` — creates all the panel items needed to display the algorithm status
- `updateElementOfInterest()` — called when the middle mouse button is pushed within a display; this can be used to display status about a particular element of region of the display
- `updateStatus()` — put current status information into the status panel items
- `statusHelp()` — provides text for a help panel that describes the contents of the status panel

The top part of the status panel in Figure 4.4 contains information about the sort algorithm. The bottom part is provided by the performance views library and displays information from each trace record as it is processed.

4.1.1.3 Algorithm Component This object is the interface between the algorithm and the rest of the system. This part of the system is not provided by AAARF since it is highly dependent on the algorithm being animated and the communications method used. These are the functions required to complete this component:

- `runAlgorithmInBackground()` — invoke the background process that executes and controls the algorithm
- `getIE()` — pass commands to and get event data from the background process

The method for executing the algorithm suggested by the AAARF manual is to implement it as a child process and get data through a UNIX socket[33] (see Figure 4.5). The parallel animation system requires a more complex implementation of this component. Since the parallel algorithm runs on another computer, the child process for this algorithm class is a program that reads data from the *data collection system* on the remote host

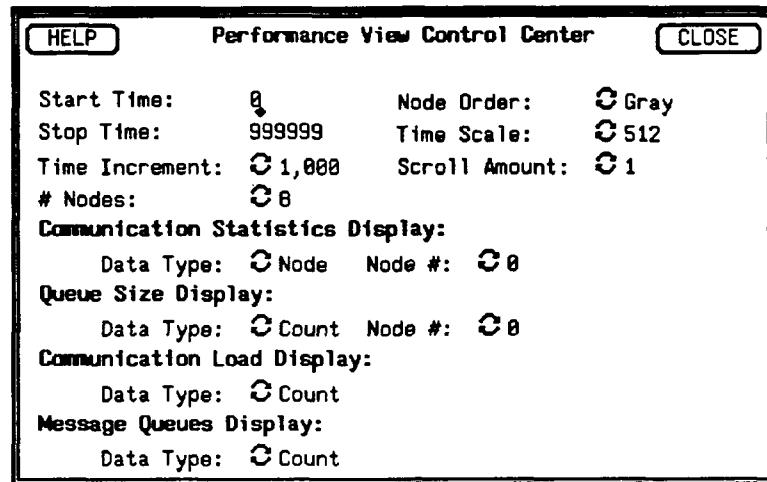


Figure 4.3. Performance Views Control Panel

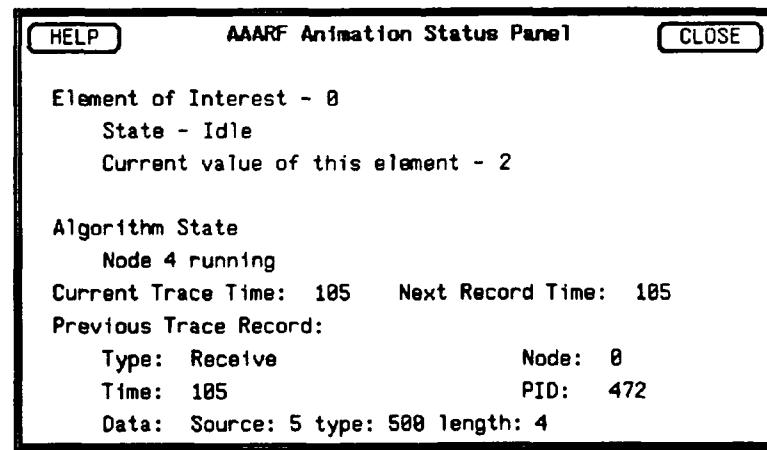


Figure 4.4. Sample Status Panel

and makes it available to the views through the UNIX socket. The child process also preprocesses the data, converting it from the PRASE format into a form more easily used by the views. The algorithm control commands and input parameters are also passed through the child process to the *algorithm control system* on the remote host. Figure 4.6 shows how the new child process interacts with the other components of the system.

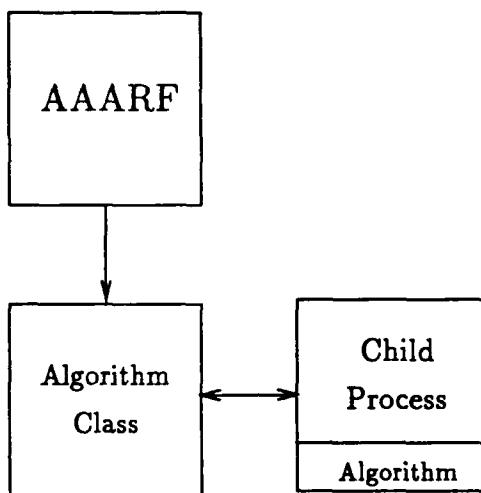


Figure 4.5. Original AAARF Child Process

The nature of the remote operation provides some conditions that cannot be ignored. A significant problem is that the remote host may not be available. The *display system* should be able to recover in the case that the target program either doesn't start, or terminates before the network connections are established. The remote operation also dictates a sequence of actions that must be followed:

1. wait for remote system to request connection for all communications links
2. establish all communications links
3. wait for and process all data
4. recognize the end of the data and shut down all links

Different things are done depending on where the current point in the sequence is. To determine what to do at a given time, the status of the remote connection is maintained;

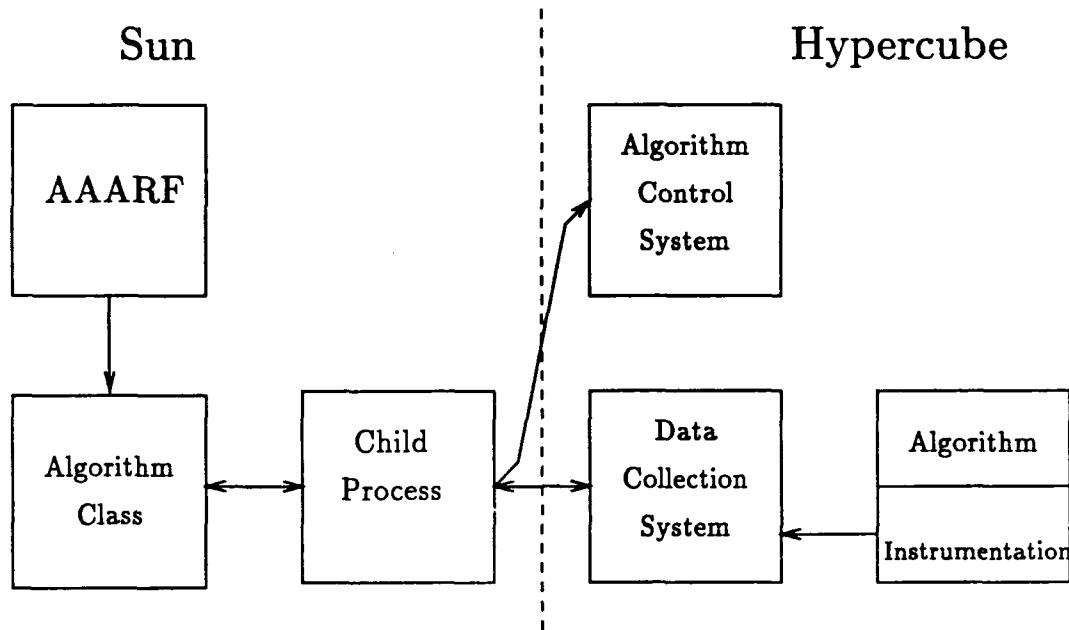


Figure 4.6. Modified AAARF Child Process

the first two steps in the sequence correspond to the remote program *starting*, item three corresponds to the remote program being *alive*, and the last step indicates that the remote program is *dead*. This sequence of activities is started every time a run is requested from the remote host.

Parallel programs tend to be very nondeterministic — two successive runs may not produce the same result; the program may not even reach the solution the same way. To give the user the capability to replay a run so that a given action can be observed without waiting for it to reoccur, the child process could write the data to disk file(s) as it arrives from the remote host. This makes three operating modes possible:

1. Live — display data as it arrives from the remote host
2. File — retrieve data from disk file(s) and display
3. Live/Write to Disk — display data as it arrives, but also write it to disk file(s) for later display

In *live* mode, the data is received from the algorithm via a network connection and translated from the PRASE data format into a local data structure. When the data is being written to disk, it is written before any translation is made, to allow the source of the data to be transparent to the rest of the processing. The translation involves performing byte-order swaps; this is required because the Intel 80386 processor stores numbers low byte first, while the Sun SPARC processor stores them high byte first[33]. The translation is accomplished on this end of the connection to avoid excessive loading of the the remote system.

A complexity introduced by the parallel animation system is that all of the event data blocks are not necessarily the same size. The approach taken by Capt Fife in the original AAARF system is to make the event block large enough to handle the largest block of data that would be sent. For local communications, this may be acceptable; but for network communications, the overhead of sending longer messages than necessary can be excessive. Since all the PRASE trace data is fixed length, one communication link can be used to send that type of data. If an event block is smaller than the length of the trace data record, it is sent using the PRASE prasemark functions. If it is longer, then a flag record is sent in the trace data link, and the data block itself is sent in another link. Since most of the data being sent is trace records, the smaller overhead involved with processing a fixed-length data stream can be exploited without incurring excessive delays by sending larger blocks than are needed.

In an attempt to avoid the situation where the view component is waiting on the child process because it uses data faster than the child process can read it from the remote system, a small buffer is used to keep records on hand. The algorithm data needs a special type of buffer — since there is no guarantee that the data is going to arrive in the exact order it is needed, there needs to be a way to make sure there is always data available (if there is any data to make available). Since each processor generates the data blocks sequentially, it suffices to make sure that at least one data block for each processor is available. In the case that more than one data block for a processor is received while a data block for another processor is being searched for, the blocks are kept in order using a queue structure, one for each processor. See Figure 4.7 for a diagram that illustrates this

process.

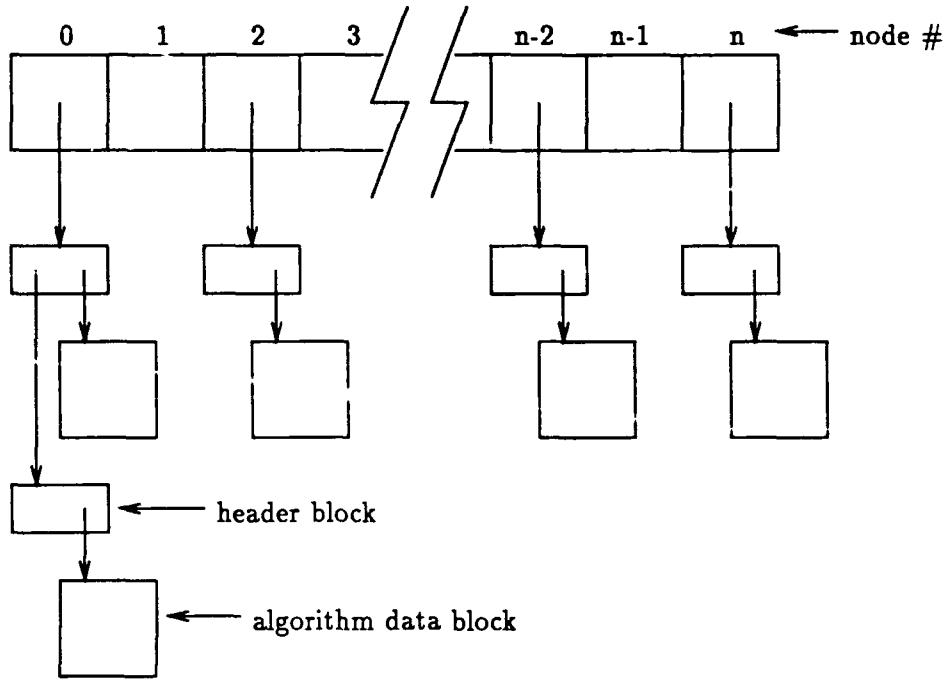


Figure 4.7. Queue Data Structure for Algorithm Data

To control the algorithm on the remote system, a two-way communications link provides the means to send commands to and receive status from the algorithm. As with the original child process management, there is a *quit* command to inform the child process to terminate. Since establishing a network connection can be a time-consuming process, this should be avoided; so instead of terminating the child process between runs in a session, a *reset* command is issued to instruct the child process to flush its queues, and to pass the *reset* command through to the remote system.

Data is retrieved from the child process by using two commands, one for trace data (the original `getIE()` function) and the other for algorithm data (a new function called `getData()`).

4.1.1.4 View Component This is the component that generates the graphics commands that are displayed in the view windows. There are three functions required to

complete this object:

- `processIE()` — processes the data in the event packet into information suitable for display
- `paintAllViews()` — generates the commands that construct the views from scratch
- `updateAllViews()` — generates the graphics commands to update each view to reflect either new data that has arrived or an advance in time

AAARF itself does not provide views — the programmer designing the animation develops custom views for each algorithm. The views may be reused with other algorithms, but the view may need to be modified to accept the data from the new algorithm.

Since most analysis of parallel algorithms involves analyzing system level performance data which does not change from one algorithm to the next, it is possible to design a set of views to display this performance data that can be reused *unmodified* by any parallel algorithm animation.

The performance views contained in the *parallel views library* show basic information regarding message passing between processes, overall communications load, communications statistics for each node, and CPU utilization. As was discussed in Section 2.5, these views are patterned after those in ParaGraph; this is by no means an exhaustive set of performance displays, but merely a collection of displays that has proven useful. The views show different types of statistics, as well as different levels of detail. The views are described in the following paragraphs:

Utilization This view is a histogram that displays the number of active processes on the vertical axis, and time on the horizontal axis. Figure 4.8 shows the utilization view during a run. The top part is colored in red to indicate the number of processors that are idle, and the bottom is green, giving the number of active processors. The *active* vs. *idle* state of a processor is derived from the message traffic: if a processor is blocked waiting for a message, then it is *idle*. Otherwise the processor is *active*.

Gantt This is a traditional view of processor activity. Each processor has a section of the vertical axis, and time is shown on the horizontal axis. When a processor is idle, that segment of the vertical axis is colored red; when the processor is active, it is green. The status of a processor is determined in the same way as for the utilization view. A sample Gantt view is shown in Figure 4.9.

Feynman The name for this view was taken directly from the corresponding ParaGraph display, though it may not have much relevance to the actual data being displayed. This view shows two different types of data: processor status and message passing activity. Figure 4.10 shows an example of this view. The processor status is shown in a similar fashion to the Gantt view, except that the data is shown as a thin horizontal line that is broken when the processor is blocked. Once again, the horizontal axis represents time. The message activity is shown by drawing lines between the horizontal lines representing the sending and receiving processors: The ends of the line are positioned according to the send and receive times. This clearly shows message passing patterns, delays, and bottlenecks. These displays are generated using trace records from send and receive system calls.

Communications Statistics This view provides a detailed look at the communications activity of a single (selectable) node. With time along the horizontal axis, the display is split into two parts: the upper half of the display shows outgoing message statistics, and the lower half shows incoming message statistics. Figure 4.11 shows a sample display. The data displayed can be chosen from three types: processor source or destination for each message, length of the messages, or the type of the messages.

Communications Load This view shows statistics on pending messages (messages that have been sent but not received) for the entire system versus time. Either the number of pending messages or the total length of all pending messages can be displayed. Figure 4.12 is an example of this view.

Queue Size Statistics for the input queue for one (selectable) node are displayed in this view. The presentation of the data is identical to the communications

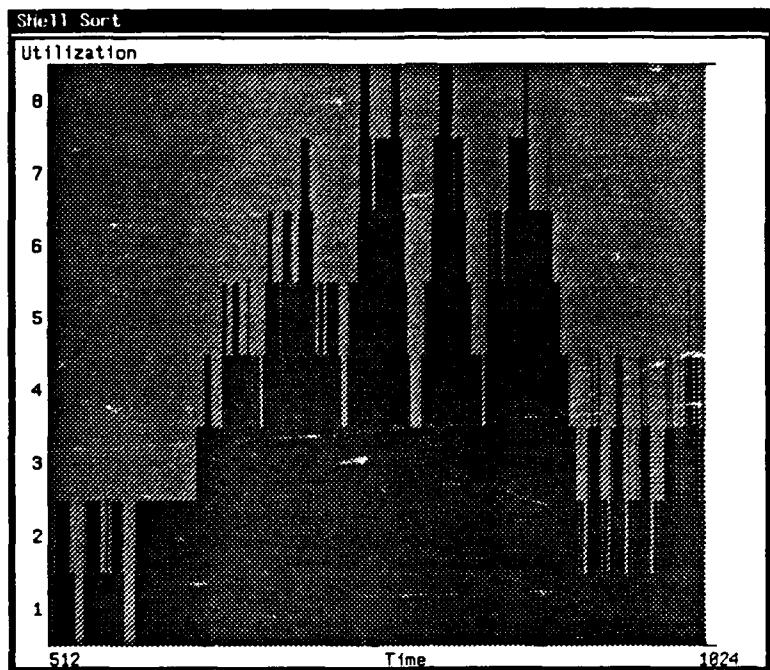


Figure 4.8. Sample Utilization View

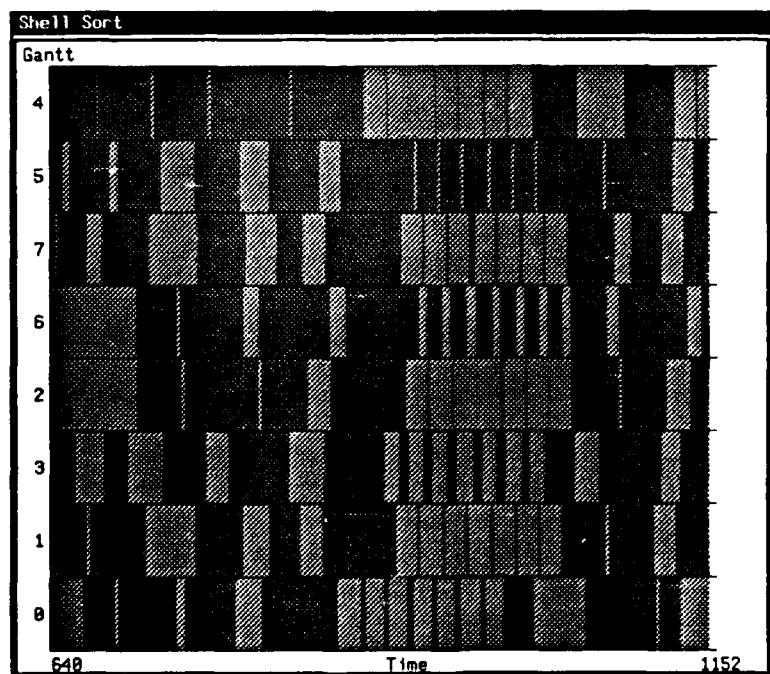


Figure 4.9. Sample Gantt View

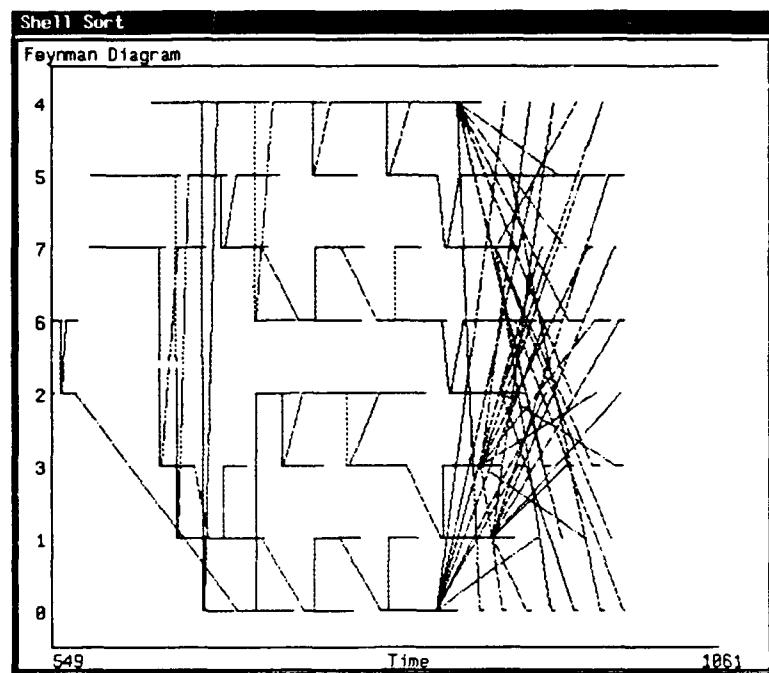


Figure 4.10. Sample Feynman View

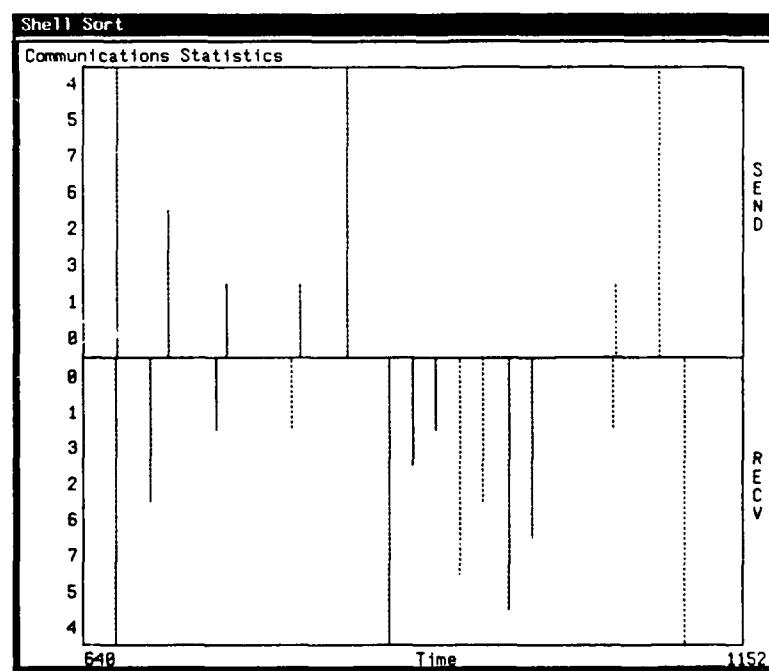


Figure 4.11. Sample Communications Statistics View

load view; The data is shown in relation to time, and either the number of messages or the total length of all messages in the input queue for the node is displayed. A sample of this view is shown in Figure 4.13.

Message Lengths This view uses a different method of presenting message passing information. The view contains a matrix, and a send operation causes an element of the matrix to be colored. The sending processor number determines the row and the receiving processor determines the column of the matrix. The length of the message determines the color used to fill in the element. See Figure 4.14 for an example.

Message Queues This view displays the current status of the input message queue for all the processors. As Figure 4.15 shows, the display is a histogram with the processors along the horizontal axis. The vertical axis can show either the number of messages in the queue or the total length of the messages in the queue. As the queue levels rise and fall, a 'high water mark' is left behind to mark the highest level that was attained for the run.

Kiviat This is very similar to a traditional kiviat chart — the processors are spread out around a circle and 'spokes' are drawn from each processor to the center of the circle (see Figure 4.16). The CPU utilization for each processor is calculated and plotted along the corresponding spoke of the wheel, with the center representing 0%. The polygon resulting from connecting points from adjacent spokes of the wheel creates a pattern that can be used to indicate the balance of the processing load among the processors. A 'high water mark' is also implemented here to indicate the highest level of processor activity.

Animation Once again, the name for this view was taken from Paragraph; the term, however, could be applied to any of the other views. This particular view shows the processors arrayed around the outside of a circle. The color of the processor indicates its status — idle, busy, blocked, or sending. In addition, lines are drawn to show message activity among the processors. When a message is sent, a line is drawn connecting the sending and receiving node; when the message is received, the line is erased. Figure 4.17 shows an example of this view.

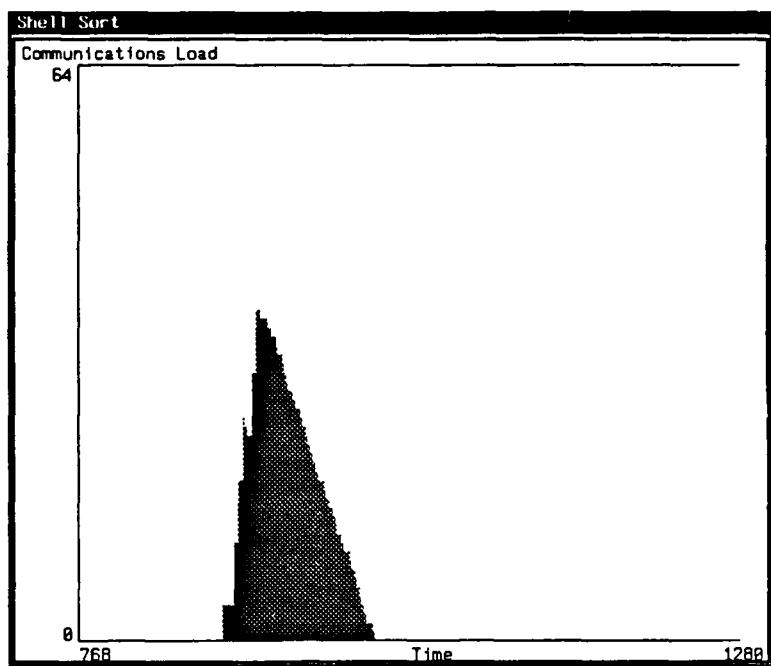


Figure 4.12. Sample Communications Load View

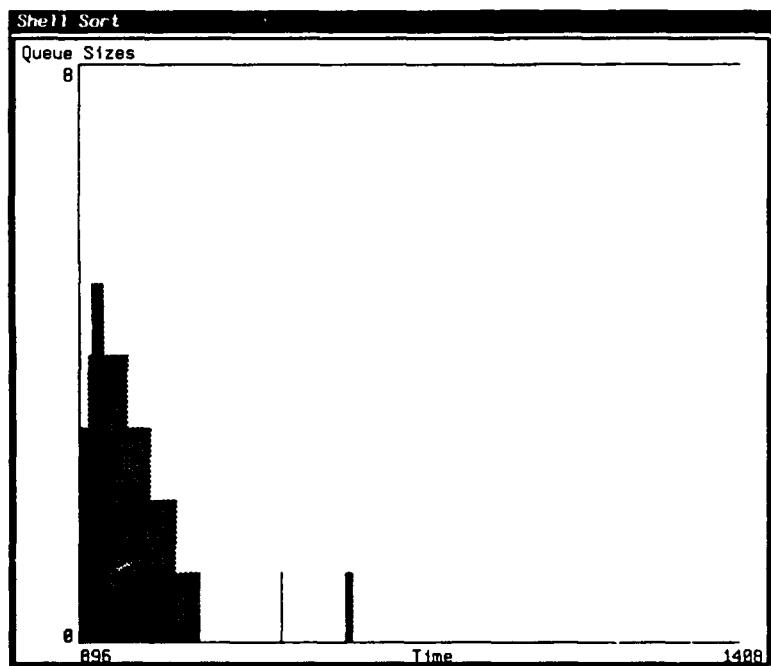


Figure 4.13. Sample Queue Size View

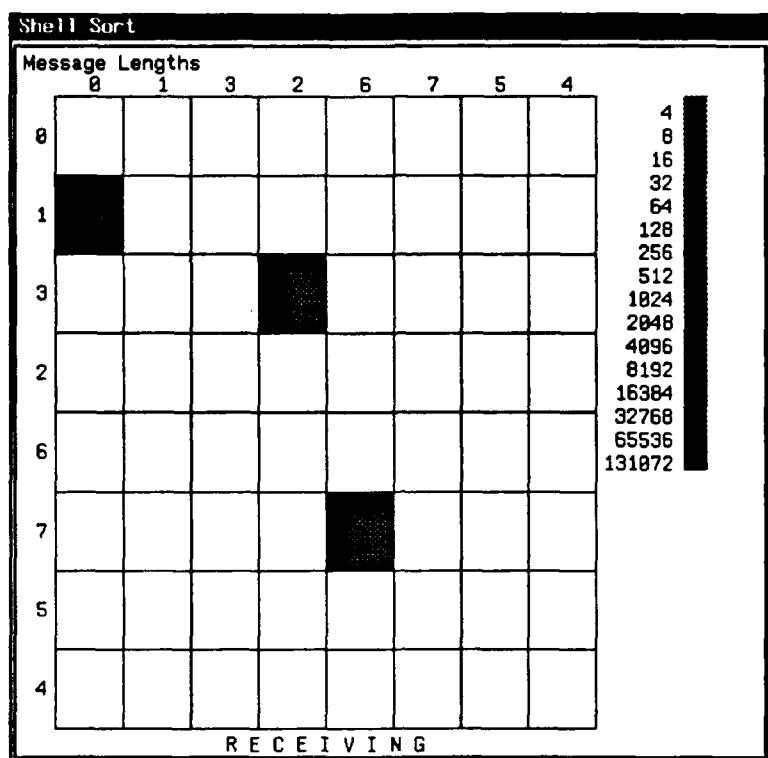


Figure 4.14. Sample Message Lengths View

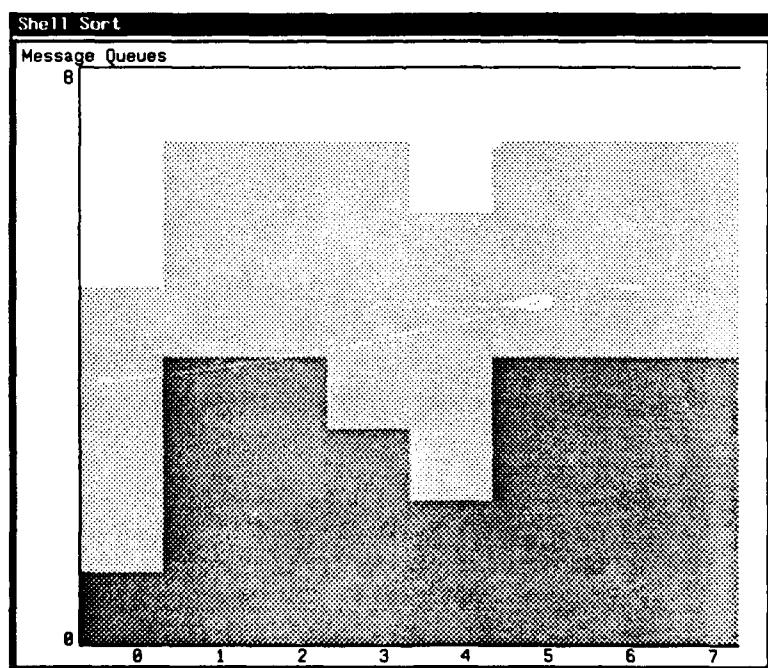


Figure 4.15. Sample Message Queues View

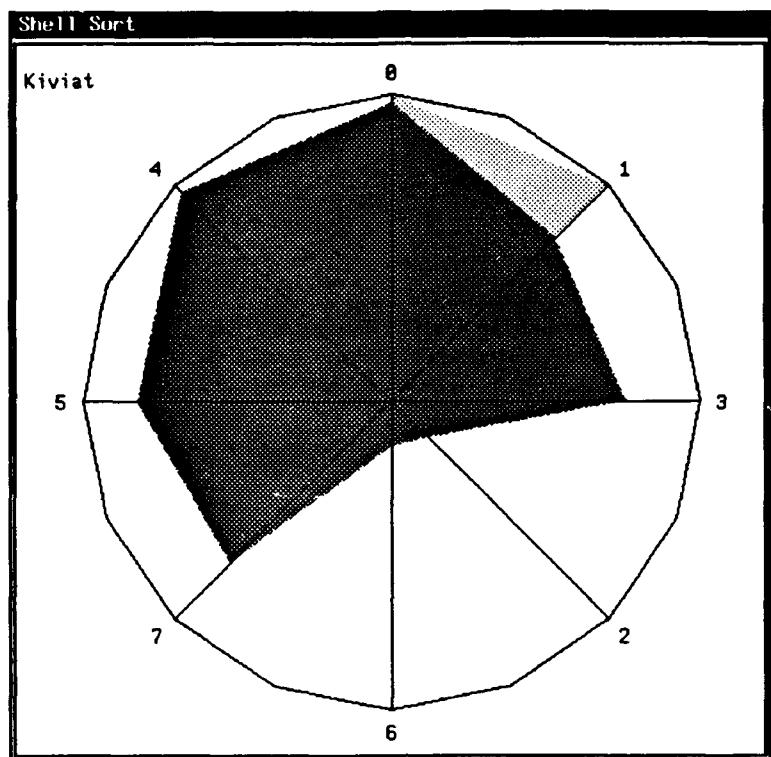


Figure 4.16. Sample Kiviat View

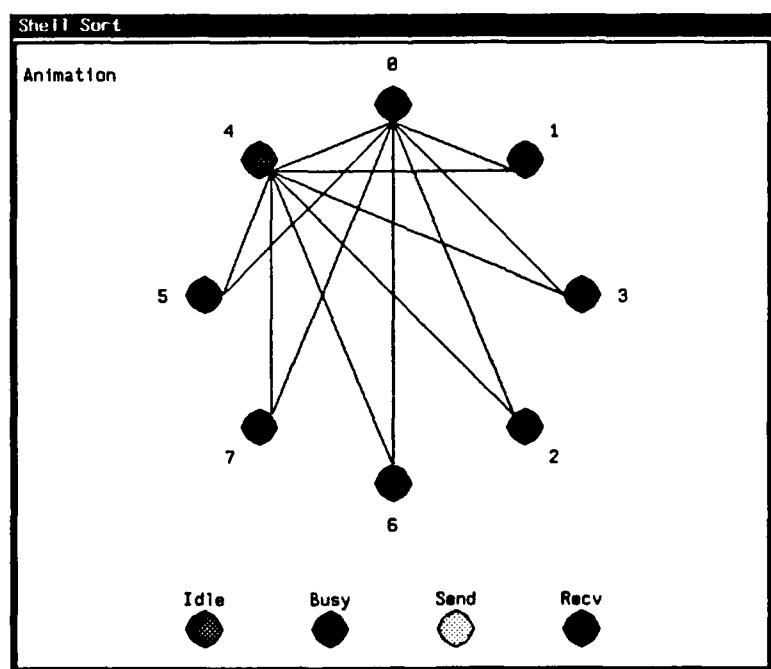


Figure 4.17. Sample Animation View

4.1.2 Modifications to AAARF AAARF needs no modification to meet the requirements in Section 2.2 or to comply with the high-level design laid out in Chapter III. Modifications are only necessary to allow the design of the algorithm class discussed in the previous section to function properly. There are only two areas that need to be changed;

1. Adapt the main processing and display cycle to allow time-based displays
2. Make the interesting event handling more general to allow the algorithm recorder to handle variable-length records

Every effort is made to prevent these modifications from affecting the use of the common library by existing sequential animations.

4.1.2.1 Time-based Displays The way in which AAARF processes algorithm events[10:39] is to get an event from the algorithm, send it to the algorithm class to process, then update the displays. This is carried out in the function `animateTheAlgorithm()` in the common library. This process is controlled by a timer interrupt inside the common library. The duration of the timer interrupt is controlled by the *Speed* slider on the *Master Control Panel* and serves to limit the rate at which events that can be processed, which slows down the update rate of the displays.

In order to implement the time-based performance views, the strictly event driven nature of the display update cycle needs to be modified to allow the timer interrupt to actually simulate an amount of time within the algorithm. Since the algorithm-specific function `processIE()` already returns a value that indicates the state of the animation, all that is needed is to pass a flag (masked onto the current return value) that indicates whether or not to get another event from the algorithm during the next cycle. During each cycle, the flag is checked and if it is set, a NULL value is passed to `processIE()` and `updateAllViews()` instead of the event. These functions recognize this and perform time-only updates instead of normal data update processing.

To prevent this modification from affecting existing animations, the flag is initialized to the FALSE state.

4.1.2.2 Generalized Event Data Handling The recording function provided by the common library assumes that the interesting event data is fixed-length and is contained in one contiguous block. This assumption fails with the data structures used in the parallel animation system.

The recording capability was designed to be transparent to all the functions that process and display event data. In the function that is executed every time the timer expires, the status of the recorder is checked, and if it is playing back a recording, the event is read from the recorder data structure rather than from the background process. This isolates the view component from the source of the data. This structure is illustrated by Figure 4.18.

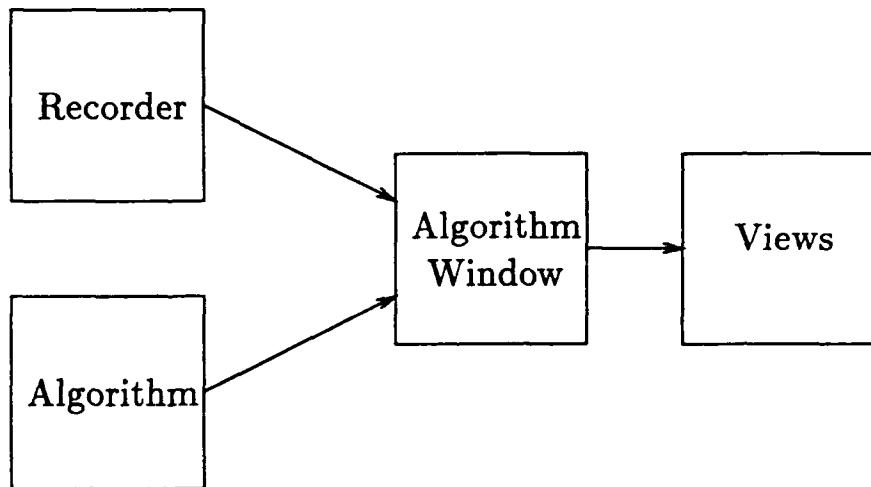


Figure 4.18. Algorithm Event Data Flow

Using an object oriented viewpoint, this is theoretically equivalent to the class structure shown in Figure 4.19. The top item is the Algorithm Window itself. This is where the decision is made as to the source of the data. The two items below it are subclasses of the Algorithm Component. Since the recorder and algorithm interface provide the same services to the algorithm window, they could be considered different implementations of the same base class. The bottom item is a new class called *Data Services*. This is the only class being discussed here that should know the exact structure of the event data. The

recorder should call upon the data services class to read and write events to a recording file. The algorithm interface class should call on the services class to read data from the child process. Isolating the structure of the event data is beneficial because none of the other classes is dependent on the structure of the data.

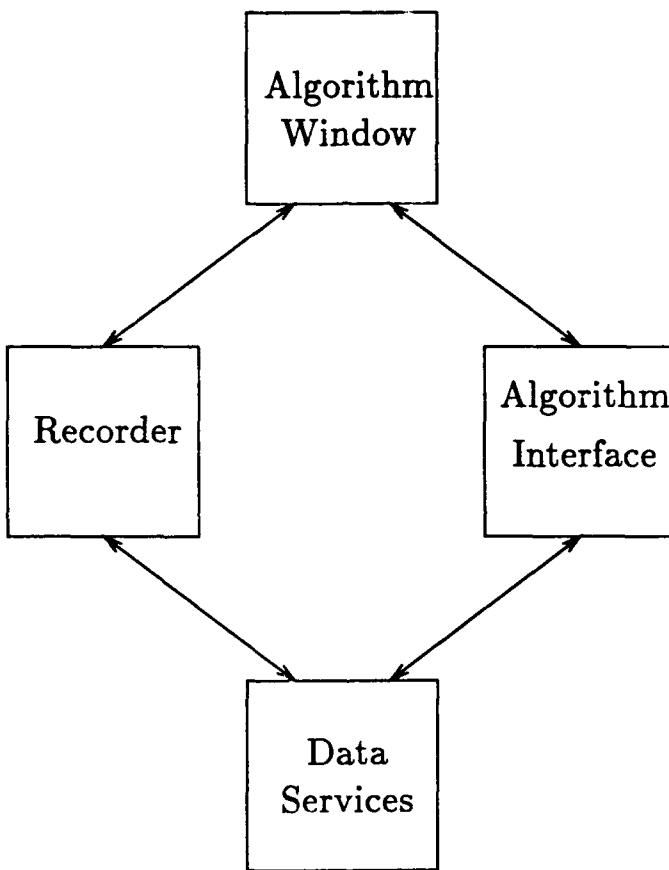


Figure 4.19. Class Structure for Recording

In the implementation of the recorder in AAARF, the class structure described in the preceding paragraph is followed, except for one item: the recorder does its own reading and writing of the event data to the recording file, instead of using the data services class. This limits the recorder's capability to save all the data. Adding two functions to the algorithm component (`readIE()` and `writeIE()`) that read and write event data to the recording file, and modifying the recorder to use these functions solves this problem and

allows the recorder to function in the parallel animation environment. The isolation of the data dependent operations is completed by adding a `freeIE()` function that destroys the event when it is no longer needed.

4.2 Data Collection System

The *data collection system* consists of two parts: the instrumentation routines that extract data from the target program, and the collection program that collects the data from the node processors and forwards it to the *display system*.

4.2.1 Instrumentation The instrumentation routines that are embedded into the target program extract trace and algorithm data from the algorithm and send it to the collection program on the host. This function is quite adequately handled by the PRASE instrumentation. The routines intercept most major system calls relating to communications and provide a means to send user-generated data along with the system trace data. It is this feature that is used to send algorithm data to the *display system*.

The instrumentation consists of three object modules that are linked in with the node programs, a header file that is included in all source files for the program, and a series of code fragments that are inserted in various parts of the program. A typical header file is shown in Figure 4.20. The main function for the node process is instrumented to initialize and terminate the tracing. Figure 4.21 shows the placement of these actions. The host program also is modified; the remote collection program is started after the cube is allocated, but before the node processes are loaded. At the end of the host program, but before the node processes are terminated, the host must wait for the remote collection program to finish collecting the data from the nodes. These modifications are shown in Figure 4.22.

4.2.2 Remote Collection Program This program has two major purposes: to relay data from the instrumented program on the processors to the *display system* and to ensure that the data from the different processors is properly interleaved to produce a time-ordered data stream to the *display system*.

```

#define PRASE

#include "aaarf_incl.h"

extern PROC prase_procs[MAX_NODES];

extern unsigned long prase_start_time;
extern unsigned long prase_lowest_node;

#define exit praseexit
#define _exit prase_exit

#define flick praseflick
#define irecv praseirecv
#define crecv prasecrecv
#define isend praseisend
#define csend prasecsend

#endif

```

Figure 4.20. Typical instrumentation header file

The PRASE program that collects trace data from the nodes and writes it to disk files, `prase_clct`, serves as the skeleton for this program. Its functionality, however, is much less than is required for the new collection program, `aaarf_clct`. As was discussed in Section 3.3.2, `prase_clct` performs no time-ordering of the data, and writes the data from each node to a separate file. `aaarf_clct` must take the data from all the nodes and produce one time-ordered output data stream. Since the modifications to `prase_clct` are so extensive, `aaarf_clct` is considered to be a new module rather than a modification to an existing PRASE program.

The instrumentation on the processors dump their trace data to the host processor by sending it via a specific message type. This means that the data from all the processors is read through one receive call by the collection program. Since there is no way to guarantee that the data being sent from different processors arrives in the same order in which it was generated, the data collection program must be able to temporarily store the data, arrange it in time order, and then send it out.

This is accomplished through the use of a prioritized linked list. As trace data arrives from the processors, it is inserted into the prioritized list according to its time stamp. The time stamp of the record at the head of the list is compared with the time of the record just received: if the difference is more than a certain threshold, the top item on the list

```

main()
{
    /* local variables */

#ifndef PRASE
    prase_procs[0].num_pids = 1;
    prase_procs[0].pids[0] = 0;
    prase_procs[1].num_pids = 1;
    prase_procs[1].pids[0] = 0;
    prase_procs[2].num_pids = 1;
    prase_procs[2].pids[0] = 0;
    prase_procs[3].num_pids = 1;
    prase_procs[3].pids[0] = 0;
    prase_procs[4].num_pids = 1;
    prase_procs[4].pids[0] = 0;
    prase_procs[5].num_pids = 1;
    prase_procs[5].pids[0] = 0;
    prase_procs[6].num_pids = 1;
    prase_procs[6].pids[0] = 0;
    prase_procs[7].num_pids = 1;
    prase_procs[7].pids[0] = 0;

    prase_lowest_node = 0;
    prase_start_time = 0;

    praseinit();
#endif

    /* normal processing */

#ifndef PRASE
    praseend();
#endif

    /* termination processing */
}

```

Figure 4.21. Instrumenting the main node function

```

main()
{
    /* local variables */

#ifndef PRASE
    FILE *prase_ptr;
#endif

    /* start of main program */
    /* initialization */
    getcube(...);

#ifndef PRASE
    system("aaarf_clct 1000000 &");
#endif

    /* load node programs */
    /* process data */

#ifndef PRASE
    praseend();
#endif

    /* terminate node programs */
}

```

Figure 4.22. Instrumenting the host process

is removed and sent to the display program. The threshold can be controlled by the user through an item on the MCP. The threshold needs to be large enough to compensate for delays in the sending of messages and for the slight difference between the clocks on the processors. It also depends on the method used to send data from the instrumentation to the host. If each trace record is sent as soon as it is generated, then the threshold value can be relatively low; but if the trace records are held up and sent in blocks, then the threshold needs to be higher to account for the time difference between the beginning and end of the block of trace records. The blocking factor for the trace records is set through a definition in the `aaarf_user.h` header file.

The algorithm data is sent to the display program as soon as it arrives, since the flag record in the trace data serves to properly order the blocks at the *display system*.

Another change from the operation of `prase_clct` is the removal of the dependence on the PRASE configuration file. `prase_clct` reads the configuration file to determine the number of `end` trace records that it needs to read before it terminates. The way the instrumentation is implemented, each process on each node sends an init trace record

when it has completed its initialization — `aaarf.clct` counts these init records and uses this count to determine when to terminate.

4.2.3 Modifications to PRASE Aside from the collection program discussed in the previous section, there are no changes to PRASE needed to meet the requirements in Section 2.2 or the system design in Chapter III. The changes described in this section are required to provide data that can be used by the performance views.

There are four things that need to be changed in PRASE:

1. Synchronization Logic
2. Timer Resolution
3. Receive-Blocking Trace Record
4. Send/Receive Matching

Each of these modifications is described in the following paragraphs.

4.2.3.1 Synchronization Logic In order to be able to display data from different nodes in the same view, the time stamps in the trace records must correspond to some global time. This is also important for matching sends with the corresponding receives. The method PRASE uses to synchronize time on the nodes is not very effective. Analysis of trace files produced for the probe effect test in Section 2.4.2.2 showed that the time stamps for some send records occurred after the time stamp for the corresponding receive record.

The method used by PRASE[19:B9-B10] is to designate the lowest numbered node as the controller, and every other node would exchange messages with it, time the messages, and attempt to correct its own time to match the controlling node. This synchronization is performed in the `praseinit()` function at the beginning of every run. This method is ineffective because the timing of the messages assumes that the controller node was waiting for the message and instantly responded with its reply. In actuality, since all nodes try to synchronize at about the same time, messages get queued up, and the timing is off.

The PICL package had no such problems because it employs an exchange of 1 byte messages with each of its nearest neighbors in the hypercube connection pattern using a blocking read. As soon as the synchronization function finishes the last read, the timer function is called to obtain the base time for the run. Analysis of PICL trace files indicates that the times on the nodes are (initially) usually within $100\mu\text{s}$ of each other.

This same synchronization method, when incorporated into the PRASE instrumentation routines, produced similar results.

4.2.3.2 Timer Resolution The 80386 processors in the Intel iPSC/2 hypercube execute some operations rapidly such that the standard `mclock()` function would produce the same time stamp for the begin and end times for each trace record; sometimes even consecutive events have the same time stamp. The `mclock()` function returns the time in milliseconds, but the time value is not necessarily updated every millisecond. This makes ordering events and calculating CPU utilization for the node processors difficult.

PICL contains a timer with a resolution of 1 microsecond. This is accomplished by directly accessing a hardware timer/counter. Normally, node programs are not allowed to directly access the node hardware, so the node program needs to be loaded with a special undocumented `xload()` function. This is unsupported by Intel and may disappear in future releases of the node operating system, but it provides the needed resolution for differentiating events during a program run. Another way to load the node program with the necessary hardware access is with the `-D` option on the load command from the command prompt[18]. This method is much better since it doesn't rely on undocumented features.

The microsecond resolution timer `pclock()` is added to the PRASE routines as a replacement for the `mclock()` system function. It returns the time in microseconds as an `unsigned long`, the same as `mclock()`, allowing a direct substitution in the source code. As a fallback, the clock routines can recognize when the node program wasn't loaded with the proper access rights, and reverts to using `mclock()`.

4.2.3.3 Receive-Blocking Trace Record Several of the performance views depend on this type of record to determine when a processor is sitting idle waiting for a message to arrive. The current PRASE only puts out one receive record when the receive has completed; the record indicates the idle period in the difference between the beginning and ending time stamps, but by the time the record is received, it is too late to reflect the idle status in the displays.

PRASE is easily modified to produce this record type by putting code to fill and send a trace record before the system blocking receive call is issued. The only data available to put in the record is the type of message being waited for, but the presence of the record in the trace is enough to properly display the node's blocked status.

4.2.3.4 Send/Receive Matching The PRASE instrumentation functions perform limited send/receive matching on the fly by assigning each message a number. Each node pair has its own sequence, which effectively counts the number of messages a given pair of nodes has exchanged. This processing is not needed in this context due to the better synchronization and higher resolution clock. The matching of sends and receives is more efficiently accomplished at the display processing stage since data from all the nodes is present. Since the processing is not needed, it should be removed to avoid maintaining unused code.

4.3 Algorithm Control System

This part of the animation receives commands directly from the *display system* and exercises control over the target program running on the remote host. It must respond to two commands from the *display system*: start the program on the remote host, and terminate the target program. These operations are accomplished using the UNIX `fork()` and `execv()`[2] to start the target program, and the `kill()` functions[2] to terminate it.

4.4 Communications

The discussion in this section is concerned mainly with the communications between the workstation running the *display system* and the remote system, as well as the com-

munication between processes on the workstation. The communication between the host and node processors on the hypercube is straightforward and is discussed in the design of PRASE[19:3-8].

4.4.1 Protocol The communications between the two computers is accomplished using UNIX *socket-based* inter-process communications[33]. The main network protocol available on systems at AFIT is the **Internet Protocol (IP)**. This protocol provides a means to directly address any host connected to the network. The **Transmission Control Protocol (TCP)** provides the mechanism to make a connection between processes residing on separate hosts.

Each side of a communications link must initialize a *socket*, specifying that the TCP/IP protocol family will be used. One side then sets up its socket into *listen* mode by selecting a port number for the TCP to use in identifying attempts to connect to the socket. Each link must choose a unique port number. After the socket is in listen mode, the process waits for the other process to initiate the connection. When the connection is terminated or broken, the socket does not need to be reinitialized; the process just waits for another connection attempt. The other side of the connection initiates the connection by specifying both the network name (or address) of the other system and the TCP port number selected by the listening side.

Once the connection is set up, the link needs no special treatment — data is passed using standard UNIX `read()` and `write()` function calls. The only significant difference between the socket and any other file or communications is that the underlying network protocols are free to take a large data block and break it up into smaller packets for transmission. If a `read()` call is issued for such a large block and all of the smaller packets haven't arrived yet, the `read()` call will return with whatever data has arrived — it doesn't wait for all the pieces to arrive. A subsequent read for the remainder of the data will receive the rest of the large block.

4.4.2 Connections Figure 4.23 shows the communications links set up for the animation system. There are two process pairs that communicate — the child process on the

workstation receives data from `aaarf.clct` on the remote host and sends back commands. The child process on the workstation also sends commands to the algorithm control process on the remote host.

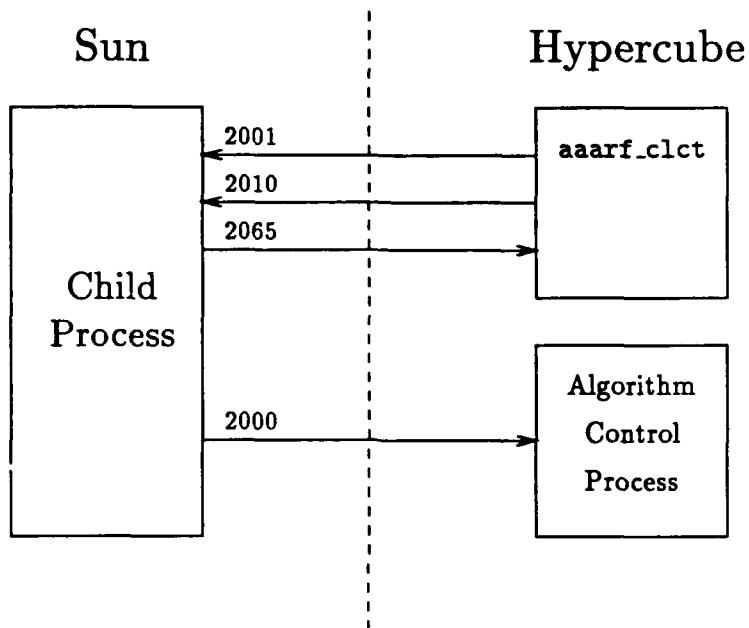


Figure 4.23. Connections between processes

The child process on the workstation receives two types of data from `aaarf.clct`: trace data and algorithm data. The simplest way to accommodate both types of data without adding overhead is to use two separate connections. The connection for the trace data uses the TCP port number of 2001, and the algorithm data connection uses port number 2010. The only command that is sent back to `aaarf.clct` is notification that all the data has been received and it may terminate. Otherwise, if `aaarf.clct` terminated before all the data has been received, the connection would be broken and the data lost. The command connection uses TCP port number 2065.

The communications from the background process and the algorithm control process only requires one link — it is a simple one-way command path. This connection uses TCP port number 2000.

4.4.3 Communications Management The background process on the workstation has several communications-related tasks:

- receive commands from the main algorithm class process
- wait for connections to be established with the processes on the remote host
- receive trace data from `aaarf_clct`
- receive algorithm data from `aaarf_clct`
- send commands to `aaarf_clct`
- send commands to the algorithm control process
- send trace data to the main process
- send algorithm data to the main process

All of the operations that send data occur in response to commands from the main algorithm class process. The tasks involving receiving or waiting can occur at any time, as is discussed in Section 4.1.1.3. There are two methods that can be used to enable the process to receive its inputs in any order. The first is to poll each input to see if data is ready using the `ioctl()` function. This method is acceptable, but the overhead of performing the `ioctl()` calls is not trivial. The other method uses another function in the UNIX library: `select()`. This function takes a list of input sources and waits until at least one of them is ready before returning. It performs the same function as `ioctl()`, but the polling loop is implemented more efficiently at a lower level. The `select()` function is versatile enough that any type of input may be monitored — both incoming data and connection attempts can be caught with the same `select()` call.

4.5 Summary

This chapter covers the detailed design and implementation of the parallel algorithm animation system. The algorithm class for AAARF is developed, as well as the *parallel views library*. Only minor modifications are made to the original AAARF system. The PRASE instrumentation routines also don't require major modification. The design of the

data collection and algorithm control processes for the remote system is discussed, and the communications system used to tie all the processes together is described.

This completes the discussion of the design of the algorithm animation system. An example of the use of this system is described in the next chapter.

V. Parallel Algorithm Animation

This chapter describes the general process of animating a parallel algorithm. As was stated in Section 1.4, it is assumed that any program can be animated; the discussion is started with an overview of parallel algorithms and programs that can benefit from animation. The rest of the chapter follows the animation of one algorithm, a parallel implementation of the NP-complete Set Covering Problem (SCP) developed by Andy Beard [3]. The target system for this example is the Intel iPSC/2 hypercube.

To support the design of other parallel animations, the methods and techniques described in this chapter are condensed into a step-by-step procedure in the updated *AAARF Programmer's Manual* in Appendix C.

5.1 Animation Process

The process of animating an algorithm can be decomposed into three tasks:

1. Analyze the algorithm to determine the basic operations
2. Develop ways of displaying the algorithm and its data
3. Instrument the program

These steps assume that the algorithm has already been implemented to the point that the program can be executed on the target system.

Even though the process has been divided into three parts, there isn't a sharp distinction between them. The analysis of the algorithm needs to be done with the displays in mind. When developing displays, the side effects of instrumentation need to be considered. Instrumentation must be done so that the displays have all the data needed so they can function. The interrelationship is so tight that all three need to be worked concurrently.

The next section discusses various categories of algorithms and how they could benefit from animation. Section 5.3 provides a very brief description of the SCP and an associated parallel implementation. For a more complete discussion of NP-Complete problems, refer to Beard [3], Aho [1], and Johnson [14]. The rest of the chapter documents the actions relating to animating the SCP.

5.2 Problem Domain

Many different types of programs can benefit from using animation. Many programs fall into two major categories: those whose execution completes in *polynomial time* (\mathcal{P} -time) and those whose execution completes in *exponential time*, if not longer (\mathcal{NP} -time or NP-hard). It should be mentioned that more complex problems also exist[1], the analysis of which could also benefit from using animation techniques.

The first category (\mathcal{P} -time) includes algorithms such as sorting, matrix manipulation, greedy searches, etc[5, 28, 17]. The parallel Shell sort implementation used for evaluating the instrumentation packages in Chapter II is an example of this class of program. The main benefit that this class of programs gets from animation is for the user or programmer to see the algorithm progressing, and to see how different methods affect the algorithm's internal processing, as well as its output.

The other class of programs (\mathcal{NP} -time) include many search techniques (especially optimal search techniques), the traveling salesman problem, and many others[14]. Since these programs have completion times that have a lower bound of $\Omega(c^n)$ time, efficient algorithms are needed to solve moderate to large problems in a reasonable amount of time. Animation can allow the programmer to easily see where the inefficiencies in the program may be for a given application. In a parallel environment, animation is also useful in showing the activity of each processor and how all the processors interact.

5.3 SCP Background

The set covering problem (SCP) is a member of the class of NP-complete problems [1:392]. The problem is to find the minimum number of columns in a 0-1 matrix such as in Table 5.1 such that each row is covered by at least one column. The columns are referred to as *sets*, hence the name. A common variation of the problem is to assign a cost to each set and find the combination of sets that both covers all the rows and has a minimum (or maximum) cost. The SCP is defined more completely by Cristofides [9:39], who also developed an *A** search algorithm for solving the problem. This algorithm is sequential in nature, but it served as the basis for Beard's work[3].

Table 5.1. Sample 0-1 Matrix

1	0	1	1	0	0	0	1	0	0	0
2	1	0	1	0	1	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	1
4	1	0	0	1	0	1	0	0	0	0
5	1	0	1	0	1	0	0	0	1	1
6	0	0	0	0	0	0	0	1	0	0
7	1	1	0	0	0	0	0	0	0	1
	2	3	6	9	4	5	1	7	3	2

The sequence of operations used by Beard are as follows:

1. Read the matrix from the data file
2. Perform all reductions that are enabled
3. Sort the matrix
4. Convert the matrix into a linked-list structure called a *Table* that the searching process uses to determine the next step to take
5. Divide the problem into pieces
6. Search until all pieces of the problem have been exhausted

The problem is divided into pieces using a data decomposition technique. The entire search process maps onto a tree structure referred to as the *search space*. The *search space* is decomposed by expanding the tree in a level manner (all the leaves are at the same depth) until the desired number of leaves exist. The search process actually works on the subtree below each leaf expanded during the division process.

Beard's implementation of the SCP distributes the searching process among the nodes of the hypercube, reserving one node to serve as a central controller. Three methods of distributing the work were implemented:

- A coarse-grained approach that divides the search space *a priori* such that each node on the hypercube knows in advance what it will search. When a node completes its work, it sits idle until the rest of the nodes complete.

- A fine-grained approach that divides the search space into more pieces than there are nodes. The controller node passes a new piece to each node as it finishes its current piece.
- A dynamic load-balancing method that starts the same as the fine-grained method, but when the controller runs out of pieces, the nodes re-distribute their workload using a token-passing technique to keep as many nodes working as is possible.

Each of the methods is (mostly) a superset of the previous one. The fine-grained method moves the division of work from the searching nodes to the controller, and the load-balancing method adds another phase of operation to the fine-grained method.

The searching process itself is a typical A^* search. The basic factor used to determine if the search should proceed along a path down the tree is the cost of the current cover — if it is greater than the current best cover maintained by the controller node, the process backtracks and tries a different path. The implementation also allows two other factors to be used in addition to the cost — a dominance test and a lower-bound test[3:3-19]. If any of the three factors indicates that the search should not proceed, that branch is abandoned and another branch is tried. The search terminates when all branches have been either searched or eliminated.

5.4 *Animating the SCP*

As was mentioned previously, the process of animating a parallel algorithm divides roughly into three major activities, but there is considerable overlap among the activities. There is also no specific order in which things must be done. It is an iterative (and concurrent) design process that only stops when the programmer is satisfied with the visualization (animation) results.

The coarse-grain version of the SCP implementation is used for this example because it provides a simple interface to the animation system. Once this animation is complete, the more advanced versions of the program could be animated by extending the initial animation.

The main area of interest when analyzing search algorithms is to look at the tree-type structure that results from the searching process, and the *state* at each node reached in the search process. This is usually referred to as the *search space*, while the organization of the data being manipulated is referred to as the *problem space*.

5.4.1 Analyzing the Algorithm In order to animate an algorithm, it needs to be analyzed to distill its basic components. Fife proposes this specification for an algorithm:

The *specification of an algorithm* can be given by the quad-tuple $\{D, I, S, E\}$ where D is a set of data structures, $I \subset D$ is a set of primary data structures, S is a sequence of instructions, and E is a set of *algorithm events*.[12:11]

Both D and I are data structures, with the difference being that while the algorithm may use the entire set of data structures contained in D , only those in I are of any interest in an “external” view of the algorithm. The set E contains those fundamental operations (or events) that, when executed in the proper order, cause the algorithm to function. These events can be input, output, or actions that change the state of the algorithm. S is the sequence in which the events are executed to accomplish the algorithm’s task.

For algorithm animation, the analysis needs to focus on D and E . S isn’t as important since the animation is showing the execution of the program, not controlling it. “Animating the algorithm consists of developing meaningful graphical representations of the algorithm events and intermediate states[12:12]”. Simply displaying the data structures and their contents is not enough since frequently (as in the SCP) algorithms don’t modify the contents of their data structures at all! The actions taken by the algorithm and the decisions made by the algorithm can present considerably more about an algorithm than just the data structures.

5.4.1.1 Data Structures The analysis of the data structures should be focused on what data is necessary to display not only the external results of the algorithm (an “external” view), but also some of the internal operations (“internal” views). In some cases, such as the SCP, the primary data structure used for display is not even used

in the algorithm! Beard's implementation uses an A^* branch and bound graph search technique[28:75-85], which is easily mapped onto a tree structure; the tree is useful because it shows the entire history of the search, including all dead-ends. The algorithm running on the hypercube uses a stack data structure to keep track of its position in the table since it has no need to maintain a history of its searching.

The significant data structures within the SCP algorithm are the matrix itself, along with the index arrays used to access it. Another data structure that is used frequently in the A^* algorithm that can be employed for animation is a simple integer array. This array is used to keep track of which sets are currently being considered for the cover.

5.4.1.2 Algorithm Events In the context of algorithm animation, there are two types of algorithm events: events that correspond to actual algorithm operations (as discussed in the previous paragraph) and events that provide information to the user or the display program. The latter type of events may include notification that a certain phase of operation is complete, or simply passing data structures to the display program to use for the displays.

The events must be chosen such that the display can be updated or modified when something "noteworthy" occurs in the algorithm. Sometimes this means choosing an event that signals the start of a process, and one that signals the end. The purpose of the event at the end may be simply to pass an updated data structure to the display, or to reset the display to keep it from getting cluttered. Events should also be identified whenever one of the significant data structures changes, which allows the displays to keep track of the current state of these structures.

The choice of events is the very core of the animation design — the rest of the animation is driven by the events produced during a run of the algorithm. These guidelines can be used to determine what events to use:

Using the program source code as implemented on the target system (or even a higher level design language), one should identify the major operations performed by the algorithm:

- Start at a high level to get the big picture, and then go to lower levels to get more detail if needed.
- Function calls in the top-level routine are prime candidates for events.
- If the program has several phases of operation, consider events that mark the transition between phases.

Remember that the level of detail needed is driven by the complexity of the displays — some displays are very effective with mainly high-level events, while others need a detailed account of the algorithm's progress.

When analyzed in this manner during the first iteration in the animation design process, the SCP program contains at least these events:

Init Cube This is an information type event that's purpose is to let the displays know that the algorithm has started, since a lot of time-consuming data transfer operations are performed at the beginning of the program.

Matrix Dims This is a data event. It provides the display with the dimensions of the matrix being searched, the number of processors allocated to do searching, and the dimension of the hypercube that is allocated for the task. This is simply a copy of a data structure that the controller processor sends to all the searcher processors.

Matrix This is another data event that sends the matrix being searched to the display. Since the matrix is actually stored in three pieces by the algorithm, this event should also be divided into three corresponding events, one for each piece.

Reduce This is an event to indicate that the program is about to enter one of the three possible matrix reductions[3:3-24]. Since the purpose of this particular animation is to display the searching process, this one event is sufficient to indicate that the program has reached this point in its execution.

Sorting Just before the searching process starts, the matrix's rows and columns are sorted to improve the efficiency of the search[3:3-9]. Once again, since the sort is not the emphasis of the animation, it is simply marked, and not expanded further.

Send Matrix Another information event whose purpose is to indicate that the final reduced and sorted matrix is being sent to the searchers by the controller.

Recv Matrix This informational event indicates that the searching processes are waiting to receive the matrix from the controller.

Forward A searcher has moved forward in the search process. The new set number and the cost of that set are included as data in the event.

Backtrack The search has run into a dead-end. The reason for the dead-end is included.

Finished Search A searcher has finished all work assigned to it and is ready to terminate.

Build Table A searcher is building the internal data structures necessary to perform the search.

Build Start Sets A searcher is determining which branches of the search tree it is responsible for searching.

New Subgraph A searcher is about to start searching a new portion of the search tree.

Finished Subgraph A searcher has finished searching a branch of the search tree.

Global Best This event marks the arrival at the controller of a solution that is the best one found so far. This is a significant occurrence since this is one factor that can limit the time spent searching for the solution.

5.4.2 Displaying the Algorithm This activity is in some ways the whole purpose of the animation effort. It is definitely the end result of the animation design process. This is also the most difficult step — there are no specific methods to generate displays for the algorithm data. There may also be several different ways that the same data can be displayed:

In an ideal program analysis environment, each aspect of . . . program behavior would be revealed by some view wherein the cause is manifest. No one view is sufficient; any particular view contains either too much or too little information to understand some aspects of a program's behavior. Unless the programmer knows exactly what to look for, interesting phenomena can be lost in the sheer volume of information.[23]

The developer of the displays relies upon creativity and experience to determine appropriate ways of displaying the data for the given application.

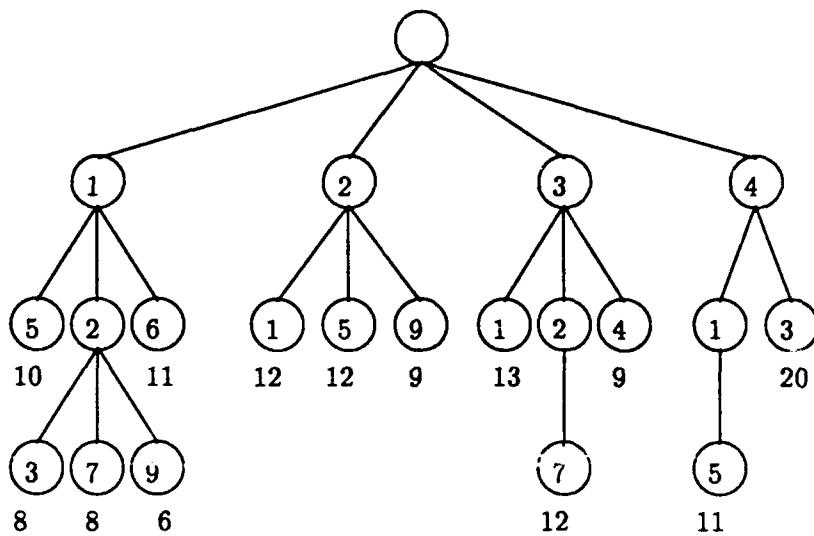


Figure 5.1. Typical representation of a SCP search tree

In the case of the SCP, one display that has already been mentioned is a representation of the search tree created by the algorithm as it progresses. A typical way of drawing the SCP search tree is shown in Figure 5.1. The information usually presented is the number of the set included in the cover at each level, possibly the cost of that set, and the total cost of the cover at each leaf of the tree. This information can easily be displayed using the data received through the events described in the previous section. In this case, there is one more dimension available: color. Although the current best cover can be represented in Figure 5.1 by using thicker lines, it is much more striking and visible to use color to show which branch of the tree is the current best cover. Another piece of information that is lacking in the normal tree is the reason that the algorithm stopped processing a branch of the search tree. The reason can be put in the figure using text and numbers, but that serves to clutter the drawing. Color is a convenient way to display the reason without unnecessarily cluttering a display that can become very crowded with a large data set.

The result of using color to enhance a traditional SCP search tree is shown in Figure 5.2. The start node at the top is colored black, and the lines connecting the nodes in the tree are black. As a node is expanded, it is colored green; when the algorithm

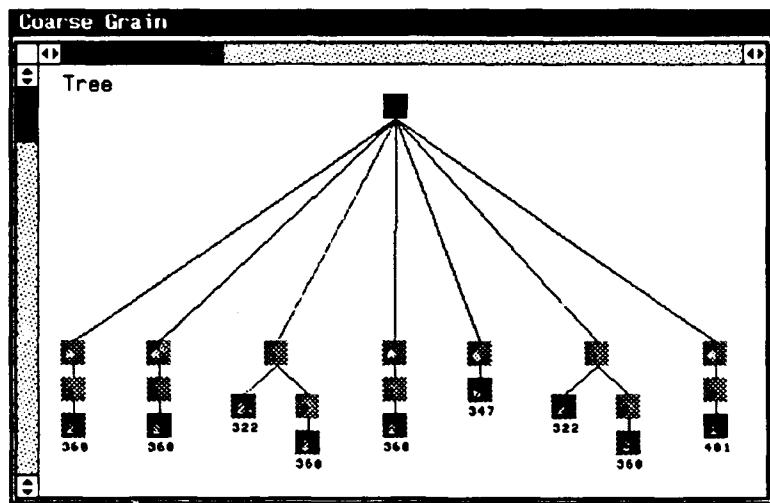


Figure 5.2. SCP tree display for a 10x10 matrix

backtracks from a node, it is colored according the reason:

Cyan the cost of advancing from that node is greater than the current best cost

Red the node represents a solution to the problem

Gray there are no more possible solutions below this node

Blue the cover so far failed the dominance test[3:3-19]

The node contains the number of the set selected at that stage in the search, and the cost for the partial or complete cover is shown below each leaf in the tree. Finally, the best cover is indicated by drawing the lines connecting the nodes in the best cover in red, and by turning the solution node at the bottom purple.

5.4.3 Instrumenting the Program Instrumenting the target program occurs in two stages — first, the performance monitoring routines are inserted, then the function calls to extract the algorithm data for the events. The performance instrumentation is inserted first for two reasons: performance data is normally needed for “complete” analysis of the algorithm, and the event data extraction functions use the performance instrumentation to carry the events to the *display system*.

Because of the existence of the PRASE preprocessor[20:3-1], inserting the performance instrumentation is simple; the programmer creates a configuration file that describes the program[20:2-3], and the preprocessor does the rest. If for some reason the instrumentation needs to be inserted manually, it is a simple process: a header file is included at the top of every source file, and some code is inserted at the very beginning and end of the main process on the node.

There are two pieces of code that need to be inserted into the host process — the `aaarf_clct` process must be started at the beginning of the run before the node processors are loaded, and the host process must wait for all the trace data to be collected before it terminates the node processes. These code fragments are described in Section 4.2.1.

Inserting the instrumentation for all the algorithm data must be done manually. As was discussed in Section 4.1.1.3, there are two methods for marking an event, depending on how much data is required to accompany the event. If there is little or no data, the `IE()` function is used; it accepts three parameters — the integer event number and two integer data values. If there is more data associated with the event than will fit in the `IE()` function call, there is another function called `IE_data()` that accepts the integer event number, a pointer to a data block, and the length of the block. The `IE_data()` function is slower, and should be used only when necessary.

For each of the events chosen in Section 5.4.1.2, one should insert one of the two event functions. There are three general situations that occur when inserting the instrumentation calls:

- The event is simply a marker, with no data
- The event requires data, and the data is already available
- The event requires data, and the data does not exist

The first case is straightforward: insert a call to `IE()` with the event number. This is illustrated in Figure 5.3. The next situation isn't as simple. If the data required for the event is in the form of one or two integers, insert a call to `IE()` with the event number and the data as parameters (see Figure 5.4). If the data is too large for `IE()`, put in a call to

IE_data() providing the event number, a pointer to the data, and the length of the data. This type of event is shown in Figure 5.5.

The last situation occurs when the event number or event data is not explicitly available from variables or location within the program. In this case, statements must be inserted into the program to determine the data or event number to send. The backtrack event for the SCP is of this type. The decision to backtrack is made implicitly by a series of "if" statements throughout the processing loop. At the end of the loop, if none of the "if" statements are true, it continues with the loop by default. The exact reason for backtracking needs to be found so that it can be passed with the event to the display. Figure 5.6 shows the code that is used to determine the reason for backtracking. For performance and efficiency reasons, this should be avoided, since the calculations needed to generate the event data can affect the operation of the algorithm; In some cases, though, it is the only way to get certain types of data.

See Appendix E for information on both the original and instrumented source for the SCP program.

5.4.4 Iteration As was stated previously, the process of animating an algorithm is an iterative activity, somewhat similar to the spiral lifecycle model for software development[4] — start out completing a small-scale implementation or prototype, then add to it in stages to develop the final implementation.

The events that are chosen depend to some extent on what is needed to create the displays; the development of the displays depends on the types of data that can be extracted without imposing an unacceptable performance penalty. The instrumentation task is the only one that truly must be done as the last step in an animation development cycle.

5.5 Ideas for Further Analysis

There are two areas for further analysis of the general SCP program: animate the more advanced stages of the program (the fine-grained and load balanced implementations), and to develop more displays for the algorithm class.

```

.
.

if( MyNID <= NumSearchers ) { /* participate in the search. */

    /* Build the table. */
    IE(BUILD_TABLE, UNUSED, UNUSED);
    PerformanceData[3] = mclock();
    if( BuildTable(&Table) == ERROR ) {
        perror("scpnodc, main(): Error building table. Exiting");
        exit(ERROR);
    }
    PerformanceData[3] = mclock() - PerformanceData[3];

    /* Calculate the starting sets. */
    IE(BUILD_START_SETS, UNUSED, UNUSED);
    PerformanceData[4] = mclock();
    TotalStartingSets = BuildStartingSets(Table, &Front, &Rear, &Current);
    Current = Front;
    PerformanceData[4] = mclock() - PerformanceData[4];

    /* Calculate the number of StartingSets for this processor. */
    PerformanceData[2] = MyNumStartingSets =
        NodeStartingSets(TotalStartingSets, NumSearchers, MyNID);

    /* Search the StartingSets. */
    NodeBestCost = MAXINT;
    PerformanceData[1] = mclock();
    PerformanceData[5] = 0L;
    for( j=0; j<MyNumStartingSets; j++ ) {
        IE(NEW_SUBGRAPH, UNUSED, UNUSED);
        MyNextStartingSet =
            NextStartingSet(Front, &Current, NumSearchers, MyNID);
        PerformanceData[5] += ScpSearch(MyNextStartingSet, &NodeBestCost);
    }
    PerformanceData[1] = mclock() - PerformanceData[1];
    IE(FINISHED_SEARCH, UNUSED, UNUSED);

    csend(FINISHED_TYPE, &Dummy, 0, CONTROL_NID, CONTROL_PID);

} /* end if( MyNID <= NumSearchers ) */

.
.

```

Figure 5.3. Event that uses no data (from `scpnodc.c`)

```

.
.

if( !DominatingSet && !A_LowerBound ) {

    /* Save this node. */
    IE(FORWARD, TheNextSet->Col, (Set+(TheNextSet->Col))->Cost);
    PushOnStack(&Top, TheNextSet);
    ExpandedNodes++;

    /* Mark the set and all vertices. */
    MarkAndAdd(TheNextSet, &NumVerticesCovered, &CurrentCover,
               &NextElement);
    CurrentCost += CostOfNextSet;
    CostOfNextSet = 0;
    if( NextElement > MaxNextElement ) {
        MaxNextElement = NextElement;
    }
} /* end if( !DominatingSet && !A_LowerBound ) */
.
.

```

Figure 5.4. Event that uses existing data (from `scpndcg.c`)

```

.
.

/* Receive AdjMat, Vertex, and Set from the host. */
ReceiveVector(MATRIX_TYPE, (char *)AdjMat, Nverts*Nsets, sizeof(int));
IE_data(MATRIX_ADJ, (char *)AdjMat, MAT_SIZE);

ReceiveVector(VERTEX_TYPE, (char *)Vertex, Nverts, sizeof(VERTEX));
IE_data(MATRIX_VERT, (char *)Vertex, VERT_SIZE);

ReceiveVector(SET_TYPE, (char *)Set, Nsets, sizeof(SET));
IE_data(MATRIX_SET, (char *)Set, SET_SIZE);
.
.

```

Figure 5.5. Event with large data (from `scpcntcg.c`)

```

.
.
.

/*****  

/* At this point, I have a cover or a I can't get a lower cost cover. */  

*****/  

/* If I have a cover and its cost < all other costs. Replace the      */  

/* best with my cover and send it to the controller.                      */  

if( NumVerticesCovered == Nverts && CurrentCost < BestCost ) {  

    BestCost = CurrentCost;  

    *(CurrentCover + NumCoveringSets) = CurrentCost;  

    csend(NEW_BEST_TYPE, CurrentCover, sizeof(int)*(NumCoveringSets+1),  

          CONTROL_NID, CONTROL_PID);  

} /* end if( NumVerticesCovered == Nverts && ... ) */  

/* IE code */  

if(NumVerticesCovered == Nverts)  

{  

    IE(BACKTRACK, SOLUTION, UNUSED);  

}  

else if(!TheNextSet)  

{  

    if(A_LowerBound)  

        IE(BACKTRACK, LOWER_BOUND, UNUSED);  

    else  

        IE(BACKTRACK, COMPLETE, UNUSED);  

}  

else if(CurrentCost+CostOfNextSet >= BestCost)  

    IE(BACKTRACK, COST, UNUSED);  

else  

    IE(BACKTRACK, UNDETERMINED, UNUSED);  

/* Backtrack. */  

do {
.
.
.

```

Figure 5.6. Event that requires derivation (from `scpndcg.c`)

5.5.1 Advanced Methods Since each method is built on the previous one, animating the more advanced SCP programs is a relatively simple task. Most, if not all, of the events chosen for the coarse-grained solution would apply to both the other methods. Just the improvements to the algorithm would need instrumentation. The load-balancing version would be the most amount of work since it adds another separate process that would need instrumenting.

5.5.2 Additional Views Only one view was developed for this example, but that doesn't mean that there aren't any other views possible. Some possible variations on the tree view developed above is to add more data concerning the state of each node. This would also make necessary a method of zooming in on a particular node of the tree; otherwise the display would get very cluttered for large data sets.

Another variation of the tree view is to have the display follow the progress of a particular processor, keeping the current position in the tree in the center of the view. This would allow the user to analyze the activities of a single processor without having to follow it manually through the whole tree.

With the implementation of the token in the load-balancing version of the SCP, another view could be developed to track the progress of the token and the load-balancing activity that it handles.

5.6 Scalability

One major difficulty in animating programs is that the amount of information required to adequately inform the user of the activities of the algorithm rapidly becomes overwhelming for medium and large sets of data. Even with high resolution displays, the minimum size of a displayed item is one pixel, and there are approximately one million pixels on a typical graphical workstation's display. Large data sets can easily create displays that exceed these display limitations — the SCP, when used to solve a 100 by 100 matrix, generates a display that exceeds the 2000 by 2000 pixel canvas allocated for the view! For smaller data sets, this is not a serious problem, but in some cases, an algorithm

behaves differently for different sized input data — displaying a large amount of data can become necessary.

One common way of displaying larger images that fit on the available display is to average adjacent pixels to reduce the number of displayed pixels without losing much of the original data. In some algorithm views, this approach may be acceptable, but for views such as the tree developed above, this approach cannot work. In general, the decision must be made between seeing every detail in the algorithm data, or seeing a “fuzzy” view that displays the algorithm at a higher level.

If a high level view is acceptable, then there must be a way to reduce the amount of data being displayed. The averaging of adjacent pixels mentioned above is one possibility if the view is a 2 or 3-dimensional graph of the data. For the tree, one possibility is to collapse the subtrees into a representative display item that still conveys the state of that part of the whole display. As the data sets get larger, statistical analysis may become useful in reducing the amount of data displayed.

Sometimes, merely having the ability to zoom in and enlarge a small section of the view, and being able to move the enlarged area around is enough to allow a display to be useful, even with very large amounts of data. Even with compression or reduction of the algorithm data to reduce the display size, the ability to zoom in to see the detail may be beneficial.

One factor that has so far been ignored in this discussion of the results of large amounts of data is the probe effect of the instrumentation. As more data is generated, the communications system can be pushed to its limit, and processes may end up being delayed because the data is being produced faster than the instrumentation and display system can process it.

5.7 *Other Applications*

To this point, the discussion about animating an algorithm has only been concerned with a full animation — analyzing the algorithm, developing specific views for the algorithm, and so forth. It is also possible that the existing view library already contains views

that adequately display the necessary information for analyzing the algorithm. There is no need to go through the effort of designing a full animation when the existing system meets the needs!

In this circumstance, the process of animating the algorithm reduces to one step — insert the performance instrumentation. An algorithm class is available for AAARF that just implements the views from the *performance views library*. It can display data from any program that contains the instrumentation. This application of the parallel animation system fills the need to simply display performance data from a program with minimal effort.

This performance display class was used to aid another student in debugging a simulation system [24]. The system was stopping unexpectedly during a run, without giving any indication as to what the problem was. The simulation system consists of different processes on each node — some generating simulation events, some consuming the events, and some nodes performed both functions. From what limited information was available, the student had concluded that there was a deadlock condition. Analysis of the performance displays revealed a different reason: the first node in the chain of producers and consumers was generating events so rapidly that the next nodes in the chain were too busy consuming the events sent to them to generate any new ones to pass along.

5.8 Summary

This chapter discussed the methods and techniques used to animate parallel algorithms. The animation of a parallel implementation of the set covering problem was described as an example of these methods. The usefulness and limitations of the animation was discussed, as well as possible improvements and additional displays. The last section discussed problems relating to animation of algorithms that operate on large amounts of data.

VI. Conclusions and Recommendations

This chapter discusses the functionality of the prototype parallel algorithm animation system developed as a part of this research. Also, conclusions that can be drawn from this research and recommendations for further research in this area are presented.

6.1 Evaluation of the Prototype

The prototype designed and implemented for this research meets the requirements stated in Chapter II:

1. Animate programs on a remote host
2. Development of built-in performance views
3. No limits on program structure
4. Display data as it is generated
5. Animate existing programs

Proper operation of the prototype was verified using three different programs written for the iPSC/2 hypercube:

- A Shell sort program
- A parallel implementation of a set covering problem (SCP)
- A simulation system

Testing of the *parallel views library* consisted mainly of black-box style tests that verified that the trace data inputs produced appropriate outputs on the display. The algorithm classes that were developed were tested in a similar fashion, using the algorithm data as the test inputs.

The three test programs were quite different in nature, which demonstrates the prototype's ability to animate a wide variety of algorithms. All three programs were developed

separately, then animated at a later time. Because of the transparency of the instrumentation, the only modifications or additions made to any of the programs were the ones required to insert the instrumentation.

Algorithm views were developed for both the Shell sort and SCP[3] programs. The Shell sort is described in Appendix A. The views for both programs were developed using views from other animations provided with the AFIT Algorithm Animation Research Facility (AAARF) package that were adapted for the data structures used in these algorithms. This demonstrates the modularity and reusability of the AAARF package. In addition, the *parallel views library* was included to provide views for performance data.

The simulation system[24] provided an additional source of performance data to test the animation system. As a result of this testing, a separate algorithm class was developed for AAARF that allows the user to view performance data from any program on the iPSC/2 that has been instrumented with the routines provided with the prototype. This algorithm class simply invokes the *performance views library* to process and display the trace data. The program may be started remotely from the master control panel for the algorithm class, or manually by a user on the hypercube. This capability allows users to view performance data without having to go through the effort of developing custom displays. It is an alternative to using systems such as Seecube[7] or ParaGraph[16] which require manual manipulation of data files that must be transferred from the hypercube to the workstation running the display.

The remote hosting of algorithms was successful, although it is possible to lock up the *display system* if for some reason the program on the remote system terminates unexpectedly. This happens because AAARF requests an event from its child process, then issues a `read()` call, which blocks until data is ready. If the algorithm has terminated, there is never going to be any data, so AAARF is locked up and must be killed from a command line in another window.

Even though the algorithm class was designed around the guidelines provided by Fife[10], one feature in AAARF does not function when a parallel animation is being run: the algorithm recorder. Section 4.1.2.2 discussed this problem and proposed a solution;

unfortunately, there was not enough time to implement this solution. This limitation is somewhat offset by the capability of the child process to write the data received from the algorithm directly into disk files. These files can then be read in at a later time to replay the run without having to rerun the program on the remote host. This also gives the capability to read and display files produced by the original PRASE instrumentation package. This 'recording' capability is limited and not as versatile as AAARF's algorithm recorder — the recorder saves the environment along with the data, allowing the view, algorithm, and input parameters to be reset to the appropriate settings when the recording is played back.

6.2 Conclusions

The prototype demonstrated that real-time animation of parallel algorithms is possible, but the speed of the various components put an upper limit on the amount of data that can be displayed. These limits are discussed in Section 6.2.1.

The design philosophy emphasizing reusability was key to the success of this research effort. By using two existing systems as a foundation, the research could focus on developing the specific areas necessary for real-time parallel algorithm animation: performance views, communications, and program instrumentation.

The methodology for animating parallel algorithms presented in Chapter V proved to be successful — the three overlapping phases of the design are sufficiently general to be applicable to any algorithm (parallel or sequential), yet are specific enough to assist in developing the animation. The only difference between animating a parallel algorithm and animating a sequential one is how to call the instrumentation function — the analysis of the algorithm and the design of the displays is the same.

The *AAARF Programmer's Manual* contained in Appendix C has been updated to cover the requirements for animating a parallel algorithm, as well as the steps that need to be taken to develop the animation. This documentation makes the task of developing the algorithm class for AAARF and instrumenting the target program very straightforward.

6.2.1 System Limitations As fast as the computer systems are today, there is still a limit on how much data may be processed. The rest of this section describes the limitations

of the components used in the prototype animation system.

6.2.1.1 Communications The communications system, while adequate for animating small-scale problems, is not fast enough to handle the massive data flow produced by an animation of a large problem. The critical link in the data flow unfortunately also is the weak link; the communications between the node and host processors on the hypercube is quite slow. This is the case with a standard iPSC/2 hypercube, which has no other method of getting data from the nodes to another system. There are hardware upgrades that allow the node processors to talk directly with remote systems, but such a system was not available for comparison.

The result of the slow communications with the host is that the outgoing message queues for the node processors fill up, eventually causing the node to block waiting to send another message. This affects the relative order in which the nodes process the problem, which can affect the outcome. There was no observed affect on the SCP other than an increase in the number of nodes explored in the search tree. This increase can be attributed to delays in each node processor receiving the global best messages being sent by the controller.

6.2.1.2 Data Reduction This limit is caused by limited CPU capacity. In the current implementation of the animation system, all the data reduction is done on the workstation. While a Sun 4 workstation has a relatively fast processor, that processor does everything — including all the data reduction and display management. Sharing the processor in this manner slows down the entire system. This is compounded by the large amount of data that is generated by large problems or detailed instrumentation.

6.2.1.3 Display Speed Aside from the communications bottleneck on the hypercube, the major slowdown results from the slow update speed of the display on the Sun 4 workstation. The time taken to redraw the active views is too long to display data in anything close to real time — for some display combinations, the screen update rate is less than once per second!

6.3 Recommendations for Future Research

There are two recommendations for further work in the area of parallel algorithm animation based on this particular investigation:

6.3.1 Animation Views There needs to be more effort invested in developing ways to display data. So far, the major effort has been to develop the tools; now that the tools are available, the emphasis should be on methods of displaying data.

The large amounts of data generated by the animation system present a major problem when displaying data — how to display the data without overwhelming the user. The use of color in animation needs to be explored further, since color can be used to convey information instead of creating a more complex display.

Currently, all the views developed for AAARF are two dimensional; the use of three dimensional graphics could be used to provide additional means to display data. This may not be feasible until the problems with display speed and CPU loading are solved — the calculations associated with displaying three dimensional data can severely load the system, possibly slowing it down too much to be useful.

6.3.2 AAARF Redesign There are two major reasons to redesign the basic AAARF program. The first is that the system is only available for the Sun workstations. While this is a common system in many academic and research facilities, there are other systems that are commonly available that could be utilized to improve the performance of AAARF.

The other major reason for redesigning AAARF is that it requires too many resources, namely window devices and memory. The SCP algorithm class with its large tree view requires 24 MB of real memory, mainly for the large canvases allocated for the views. This also limits the number of algorithm classes that can be active simultaneously to one! The number of window devices required when four algorithm classes are active exceeds 128. These resource requirements exceed the available resources on most available workstations — many Sun workstations are only configured with 8 MB of memory and 64 window devices. The additional window devices can be created, but it adds additional overhead in the UNIX kernel. The redesign process needs to rework the methods used to allocate

resources so that they are only allocated when needed, instead of allocating all resources at initialization.

These two problem areas, as well as the CPU loading and possibly the display speed problems could be helped considerably if the redesign included a new implementation under the X window system[29:354]. X allows, even encourages, portability among different hardware environments. This permits AAARF to be designed to run acceptably on a minimally capable system, yet allow it to benefit from higher performance systems.

The resource allocation methods can be reworked during the redesign, but the effort can benefit from X's different handling of resources — each item in a display under X doesn't require a specific entry in the device tables. Each view can be contained in a separate window, allowing it to be allocated and deallocated on demand.

One major benefit of using X is that it has built-in support for distributing the processing and display activities over several systems connected by a network. This can be used to ease the load on the system driving the display. This way, the different components can be placed on systems that are best suited to the task. The use of X can also help ease the display speed problem by utilizing multiple displays on different systems; one X client can connect to multiple X display servers.

6.4 Summary

The methodology presented in Chapter V proved to be useful in developing animations. The development of a system that can animate parallel algorithms in real time was successful, but several problems were discovered. The major problem is speed; both processing speed and communications speed. Two recommendations are made — put more effort into developing displays for animations and redesign AAARF for use with the X window system.

Appendix A. *Parallel Shell Sort*

This appendix contains the source for the Shell sort used to evaluate the instrumentation packages in Chapter II and to test the animation system. The program was developed as a lab exercise for the EENG 689 *Algorithms for Parallel Processing* course. The algorithm was derived from one found in *Solving Problems on Concurrent Processors*[13:341], one of the texts used for the course.

The program consists of two source files, one for the program on the host processor that distributes the data to the nodes and collects it at the end of the run, and the other is the program that runs on all the node processors. The node program was instrumented with the PRASE C preprocessor.

```
/* -----> Begin Code added by the PRASE C Preprocessor */
```

```
#include "prase_extern.h"
```

```
#define exit praseexit
#define _exit prase_exit
```

```
#define crecv prasecrecv
#define csend prasecsend
```

10

```
/* -----> End Code added by the PRASE C Preprocessor */
```

```
*****
```

```
* FILE : Shell.c
* DATE : 1 November 90
* VERSION: 1.1
* AUTHOR : Capt Ed Williams
* DESCRIPTION:
```

```
* This program runs on each of n nodes.
```

20

```
* FUNCTIONS:
```

```
* main() - Directs execution of the Shell sort.
* Ascend() - Comparison routine for qsort
* merge_hi() - performs merge with another node and keeps the
* high part of the result
* merge_low() - performs merge with another node and keeps the
* low part of the result
* msort() - performs a merge sort
```

```
* HISTORY:
```

30

```
* Borrows heavily from the bitonic sort which was written
* by K Fife and M Proicou.
```

```
* (1 May 90) version 1.0 - initial version for EENG 689 Lab
```

```
*****
```

```
#include <stdio.h>
```

```
#include "aaarf.h" /* to get IE definitions */
```

```
/*
** cube node functions
*/
```

40

```
long ginv(), gray(), infocount(), infonode(), infopid(), infotype();
long iprobe(), irecv(), isend(), myhost(), mynode(), mypid(), nodedim();
long numnodes();
long mclock();
```

```
#define MANAGER myhost()
#define MY_HOST_PID 1L
#define NODE_PID 472L
#define ALLNODES -1
#define PKT_DATUM 51L
#define PKT_LENGTH 50L
#define STATUS 530L
```

myhost

50

```

#define PKT_SENDA      128L
#define PKT_SENDB      0L
#define WORK_SIZE      65536L
#define NODE_PID       472L

int      Data_area[WORK_SIZE], Work_area[2*WORK_SIZE];
long     my_pid;
long     my_node_id;
int      chunk_size, chunk_len;

/* ****
 * FUNCTION:      main()
 * FUNCTION NUMBER: 0 (Shell.c)
 * VERSION NUMBER: 1.0
 * DATE:          1 May 90
 * DESCRIPTION:
 *      Runs the Shell Sort. Accepts data length and the data
 *      from the host, sorts its piece of the data, then starts
 *      the two-stage sort process. The nodes are organized as
 *      a gray-code ring. The first stage is a diminishing
 *      increment merge. The second stage is a "mop-up"
 *      phase which uses a nearest neighbor merge repeatedly
 *      until the data is completely sorted.
 * INPUT PARAMETERS:
 *      piece of data from host
 * OUTPUT PARAMETERS:
 *      piece of sorted array to host
 * CALLED BY:
 *      Loaded by host program ShellHost
 * FUNCTIONS CALLED:
 *      merge_hi(), merge_low(), qsort()
 * HISTORY:
 *      (1 May 90) version 1.0 - initial version for EENG 689 Lab
*****/
```

main() {

```

int Ascend();
```

char buf[80];

int i, done, other_status;

long num_nodes, mask, ring_pos;

long cnt, fr_node, fr_pid;

long d;

long time;

/*#\$*PRASE BEGIN MAIN*/

/* -----> Begin Code added by the PRASE C Preprocessor */

```

prase_procs[0].num_pids = 1;
prase_procs[0].pids[0] = 472;
prase_procs[1].num_pids = 1;
```

main(Shell.p.c)

```
prase_procs[1].pids[0] = 472;
prase_procs[2].num_pids = 1;
prase_procs[2].pids[0] = 472;
prase_procs[3].num_pids = 1;
prase_procs[3].pids[0] = 472;
prase_procs[4].num_pids = 1;
prase_procs[4].pids[0] = 472;
prase_procs[5].num_pids = 1;
prase_procs[5].pids[0] = 472;
prase_procs[6].num_pids = 1;
prase_procs[6].pids[0] = 472;
prase_procs[7].num_pids = 1;
prase_procs[7].pids[0] = 472;                                110

prase_lowest_node = 0;
prase_start_time = 0;                                         120

praseinit();
```

```
/* -----> End Code added by the PRASE C Preprocessor */
```

```
my_node_id = mynode();          /* my node number */
my_pid     = mypid();
d          = nodedim();         /* Dimension of the cube */
num_nodes  = 1 << d;                                130

/*
** Find out how many elements to sort
*/
crecv(PKT_LENGTH, &chunk_size, sizeof chunk_size);
if(chunk_size > WORK_SIZE)                                         140
{
    printf("Can't handle array of size %d, max is %s\n",chunk_size,WORK_SIZE);
}

/*
** Messages with chunk_size elements, require chunk_len bytes
*/
chunk_len = chunk_size * sizeof(int);

/*
** Get the data to be sorted from the cube manager
*/
crecv(PKT_DATUM, Data_area, chunk_len);
IE(GOT_DATA, 0, 0);
IE_data(HIDE, Data_area, chunk_len);
time = mclock();                                         150

/*
** sort the data, so that only merges are required later
*/
```

main(Shell.p.c)

```

*/
IE(IND_SORT, 0, 0);
qsort(Data_area, chunk_size, sizeof(int), Ascend);
IE_data(HIDE, Data_area, chunk_len);

/*
** this is the decreasing increment sorting process
**
** each loop cuts the distance between nodes in half
*/
for(i = d-1; i >= 0; i--)
{
    mask = 1 << i;
    if(ginv(my_node_id) > ginv(my_node_id ^ mask))
        merge_hi(my_node_id ^ mask);
    else
        merge_low(my_node_id ^ mask);
}

/*
** this is the mop-up stage
**
** it is similar to a parallel bubblesort, except instead of
** compare/swap operations, the standard merge operation is
** used.
*/
done = 1;
while(done)
{
    ring_pos = ginv(my_node_id);
    if(ring_pos == 0)
    {
        done = merge_low(gray(ring_pos + 1));
    }
    else if(ring_pos == num_nodes - 1)
    {
        merge_hi(gray(ring_pos - 1));
        done = 0;
    }
    else if(ring_pos & 1)
    {
        merge_hi(gray(ring_pos - 1));
        done = merge_low(gray(ring_pos + 1));
    }
    else
    {
        done = merge_low(gray(ring_pos + 1));
        merge_hi(gray(ring_pos - 1));
    }
}

/*
** if done is still 0, then no changes were made.
** signal the rest of the nodes
*/

```

main-Ascend(Shell.p.c)

```
/*
IE(SEND_STATUS, 0, 0);
csend(STATUS, &done, sizeof(int), ALLNODES, NODE_PID);

/* wait for all nodes to reply */
IE(WAIT_STATUS, 0, 0);
for(i = 0; i < num_nodes-1; i++)
{
    crecv(STATUS, &other_status, sizeof(int));
    done += other_status;
}
IE(RUNNING, 0, 0);
}

/*
** Send my portion of sorted data to host
*/
time = mclock() - time;                                230
IE(SEND_DATA, 0, 0);
csend(PKT_DATUM, Data_area, chunk_len, MANAGER, MY_HOST_PID);

printf("Sort took %d ms on node %d\n", time, my_node_id);

/* -----> Begin Code added by the PRASE C Preprocessor */
praseend();                                              240

/* -----> End Code added by the PRASE C Preprocessor */

}

*****  

* FUNCTION:      Ascend(a, b)
* FUNCTION NUMBER: 1 (Shell.c)
* VERSION NUMBER: 1.0
* DATE:          1 May 90
* DESCRIPTION:
*      comparison function for use by the quicksort library routine.
* INPUT PARAMETERS:
*      a, b - pointers to data items
* OUTPUT PARAMETERS:
*      Comparison result
* CALLED BY:
*      qsort()
* FUNCTIONS CALLED:
*      none
* HISTORY:
*      (1 May 90) version 1.0 - initial version for EENG 689 Lab
*****/
```

int Ascend(a,b)

Ascend

```

int *a,*b;
{
    if(*a > *b)
        return 1;
    if(*a < *b)
        return -1;
    return 0; /* equal */
}

```

270

...myhost

```

*****+
* FUNCTION:      merge_hi(dest)
* FUNCTION NUMBER: 2 (Shell.c)
* VERSION NUMBER: 1.0
* DATE:          1 May 90
* DESCRIPTION:
*   Merge with other node and keep the high part of the
*   data. This is done by sending this node's data to
*   the other node, who does the merge, and wait for the
*   high data to come back.
* INPUT PARAMETERS:
*   dest - node to merge with
* OUTPUT PARAMETERS:
*   none
* CALLED BY:
*   main()
* FUNCTIONS CALLED:
*   none
* HISTORY:
*   (1 May 90) version 1.0 - initial version for EENG 689 Lab
*****/

```

280

290

merge_hi

```

merge_hi(dest)
long dest;
{
    int i, fr_node, fr_pid, cnt;
    char buf[81];
    /*
    ** Send data to destination node
    */
    IE(MERGE_HI, dest, 0);
    csend(PKT_SENDA+my_node_id, Data_area, chunk_len, dest, NODE_PID);

    /*
    ** Wait for data from other node
    */
    crecv(PKT_SENDB+(dest), Data_area, chunk_len, &cnt, &fr_node, &fr_pid);
    IE_data(HIDE, Data_area, chunk_len);
}

```

300

310

...myhost

```

*****+
* FUNCTION:      merge_low(dest)

```

myhost-merge_low(Shell.p.c)

```

* FUNCTION NUMBER: 3 (Shell.c)
* VERSION NUMBER: 1.0
* DATE: 1 May 90
* DESCRIPTION:
*   Merge with the other node and keep the low part of the
*   data. This is done by waiting for the other node to
*   send its data, then performing the merge, then sending
*   the high end of the data back to the other node.
* INPUT PARAMETERS:
*   dest - node to merge with
* OUTPUT PARAMETERS:
*   none
* CALLED BY:
*   main()
* FUNCTIONS CALLED:
*   msort()
* HISTORY:
*   (1 May 90) version 1.0 - initial version for EENG 689 Lab
******/
```

merge_low(dest) 320

long dest;

{

int i, fr_node, fr_pid, cnt; 340

char buf[81];

int done_flag;

/* Receive from node (dest) */

IE(MERGE_LOW, dest, 0);

crecv(PKT_SENDA+(dest), Work_area+chunk_size, chunk_len);

/*

** At this point:

** -----

** Work_Area = | empty | Stuff Just Read | 350

** -----

**

** Merge stuff just read with my data area. Stuff is

** already sorted so only a merge is needed. Merge places

** its result into Work_area.

**/

done_flag = msort(Work_area+chunk_size, Data_area, chunk_size,

chunk_size, Work_area);

/*

** move low data back into local data area 360

*/

for(i = 0; i < chunk_size; i++)

Data_area[i] = Work_area[i];

/*

** send hi data back to other node

*/

csend(PKT_SENDB+my_node_id, Work_area+chunk_size, chunk_len,

dest, NODE_PID); 370

merge_low-msort(Shell.p.c)

```
IE_data(HIDE, Data_area, chunk_len);

    return done_flag;
}

/***** myhost *****
* FUNCTION:      msort(iptr, jptr, sizeof_i, sizeof_j, tArray)
* FUNCTION NUMBER: 4 (Shell.c)
* VERSION NUMBER: 1.0
* DATE:          1 May 90
* DESCRIPTION:
*   performs a merge sort
* INPUT PARAMETERS:
*   iptr, jptr - pointers to data blocks
*   sizeof_i, sizeof_j - sizes of each data block
*   tArray - pointer to destination area
* OUTPUT PARAMETERS:
*   merged array
* CALLED BY:
*   merge_low()
* FUNCTIONS CALLED:
*   none
* HISTORY:
*   (1 May 90) version 1.0 - initial version for EENG 689 Lab
***** msort *****
msort(iptr, jptr, sizeof_i, sizeof_j, tArray)
int *iptr, *jptr, *tArray, sizeof_i, sizeof_j;
{
    int i = 0, j = 0, k;
    int flag;
    int *kptr, *tiptr, *tjptr;

    kptr = tArray;
    tiptr = iptr;
    tjptr = jptr;

    while(i < sizeof_i && j < sizeof_j)
    {
        if(*iptr < *jptr)
        {
            *kptr++ = *iptr++;
            i++;
        }
        else
        {
            *kptr++ = *jptr++;
            j++;
        }
    }
    flag = !(i == 0);
    if(i == sizeof_i)
    {
        for(; j < sizeof_j; j++)

```

```
msort(Shell.p.c)

    *kptr++ = *jptr++;
}
else
{
    for(; i < sizeof_i; i++)
        *kptr++ = *iptr++;
}
return flag;
}
430
```

ABS(ShellHost.c)

ABS-main(ShellHost.c)

```
/*
int sortedness = 0;
int seed = 0;
int pattern = 0;
int cycles = 1;
int invert = 0;

/*
** variables used to set the cube
*/
char CubeName[8]. Nodes[5];
int Status, NextName, Waiting = FALSE;
unsigned long WaitTime;

*****  

* FUNCTION:          main()
* FUNCTION NUMBER:  0 (ShellHost.c)
* VERSION NUMBER:  1.0
* DATE:            1 May 90
* DESCRIPTION:
*     Inputs the number of elements to sort from the command
*     line, starts the sort, and then checks the result.
* INPUT PARAMETERS: none
* OUTPUT PARAMETERS: none
* GLOBALS USED: value - data array
* GLOBALS AFFECTED: none
* FUNCTIONS CALLED: do_Shell() - Shell sort driver
*                     CheckSort() - checks that array is actually sorted
* HISTORY:
*     (1 May 90) version 1.0 - initial version for EENG 669 Lab
*****/  

main(argc, argv)
int argc;
char *argv[];
{
    unsigned items;
    unsigned long num_nodes;
    if (argc < 3 || argc > 8)
    {
        printf("Run and time a parallel Shell sort on a hypercube \n");
        printf("  USAGE: ShellHost number_of_elements number_of_nodes\n");
        printf("          number_of_elements - between 32 and 65536\n");
        printf("          number_of_nodes - nodes to use\n");
        exit(1);
    }
    else if(argc == 3)
    {
        items = atoi(argv[1]);
        num_nodes = atoi(argv[2]);
    }
    else if(argc == 8)
        main(argc-2, argv+1);
    else
        items = 32;
    num_nodes = 65536;
    sortedness = 0;
    seed = 0;
    pattern = 0;
    cycles = 1;
    invert = 0;
    Status = FALSE;
    NextName = 0;
    Waiting = FALSE;
    WaitTime = 0;
    CubeName[0] = 'S';
    CubeName[1] = 'h';
    CubeName[2] = 'e';
    CubeName[3] = 'l';
    CubeName[4] = 'l';
    CubeName[5] = 'o';
    CubeName[6] = 's';
    CubeName[7] = 't';
    Nodes[0] = '0';
    Nodes[1] = '0';
    Nodes[2] = '0';
    Nodes[3] = '0';
    Nodes[4] = '0';
    Nodes[5] = '0';
    Nodes[6] = '0';
    Nodes[7] = '0';
    do_Shell(items, num_nodes, sortedness, seed, pattern, cycles, invert, Status, NextName, Waiting, WaitTime, CubeName, Nodes);
    if (Status)
        CheckSort(items, num_nodes, sortedness, seed, pattern, cycles, invert, Status, NextName, Waiting, WaitTime, CubeName, Nodes);
    if (Status)
        printf("Sort successful\n");
    else
        printf("Sort failed\n");
    exit(0);
}
```

main(ShellHost.c)

```
{  
    items = atoi(argv[1]);  
    num_nodes = atoi(argv[2]);  
    sortedness = atoi(argv[3]);  
    seed = atoi(argv[4]);  
    pattern = atoi(argv[5]);  
    cycles = atoi(argv[6]);  
    invert = atoi(argv[7]);  
}  
    sprintf(Nodes, "%d", num_nodes);  
    do {  
        sprintf(CubeName, "Shell%d", NextName);  
        Status = _getcube(CubeName, Nodes, NULL, 0);  
        if(Status == -1)  
        {  
            if(errno == EQNOCUBE)  
            {  
                if(!Waiting)  
                {  
                    printf("%d node(s) not available, retrying ...\\n", num_nodes);  
                    Waiting = TRUE;  
                }  
                WaitTime = mclock();  
                while( mclock()-WaitTime < 10000.0 )  
                ;  
            }  
            else if( errno == EQCUBEEEXISTS )  
            {  
                ++NextName;  
            }  
            else  
            {  
                perror("Error during _getcube(), Exiting");  
                exit(1);  
            }  
            /*Status = _getcube(CubeName, sNodes, NULL, 0);*/  
        }  
        while( Status == -1 );  
        printf("Allocated cube is called %s.\\n", CubeName);  
        setpid(HOST_PID);  
        system("aaarf_clct &");  
        if(items > (MAX_ARRAY_SIZE))  
        {  
            printf("Sorry, can only sort %u items.\\n", MAX_ARRAY_SIZE);  
        }  
        else if(items < 32)  
        {  
            printf("Sorry, need at least 32 items to sort. \\n");  
        }  
        else  
        {  
110  
120  
130  
140  
150
```

main-do_Shell(ShellHost.c)

```
    do_Shell(value, items);
}
exit(0);
}

*****  
* FUNCTION:      do_Shell(values, num_items)
* FUNCTION NUMBER: 1 (ShellHost.c)
* VERSION NUMBER: 1.0
* DATE:          1 May 90
* DESCRIPTION:
*   Does a Shell merge sort on an Intel hypercube. The node program
*   Shell is loaded, and the data is partitioned among the processors
*   for sorting. This procedure generates a vector, using storage
*   space allocated by the main routine and passed as a parameter.
* INPUT PARAMETERS:
*   A pointer to a vector of integers (values) and the
*   number of items to put in the vector.
* OUTPUT PARAMETERS:
*   The vector is filled with numbers, and sorted.
* CALLED BY: main()
* HISTORY:
*   (1 May 90) version 1.0 - initial version for EENG 689 Lab
*****/  
  
#define PKT_LENGTH_MSG 50      /* Packet type for length msg */
#define PKT_DATUM_MSG 51      /* Packet type to send/receive sorted data */
#define MAX_ARRAY MAX_ARRAY_SIZE /* Maximum Size of array to sort */  
  
#define LF_START 1
#define LF_UNPROT 4  
  
do_Shell(values, num_items)
int values[];
int num_items;
{
    long num_nodes,      /* number of active nodes in the cube */
        cnt,           /* number of bytes received */
        fr_node,        /* node id originating message */
        fr_pid;        /* process id originating message */
    int i;              /* counter */
    extern int rand();    /* Random number generator */
    extern char *malloc();
    int chunk_size, chunk_len;
    long node_idx;
    int j, all_nodes = -1;
    long ring_pos;
    void randarray();  
  
    int *sorted_data; /* pointer to return area of sorted data */
  
/* Start of code added for PRASE routines */  
160
170
180
190
200
210
...ABS
```

```

FILE *prase_ptr;

/* End of code added for PRASE routines */

/*load("Shell", ALL_NODES, PID);*/
_xload("Shell", &all_nodes, 1, PID, LF_START | LF_UNPROT, 0);
num_nodes = 1<<nodedim();
if((num_items/num_nodes)*num_nodes != num_items) 220
{
    puts("Error, number of items must be a multiple of the number of nodes");
}

randarray(values, seed, sortedness, num_items, pattern, cycles, invert);

/*
** uncomment the appropriate statement for the data organization
*
for (i = 0; i < num_items; i++) 230
{
    values[i] = rand();
    values[i] = i;
    values[i] = num_items - i - 1;
}
*/

/*
** Tell the node program the size of the chunk it must sort
** chunk_size == Number f elements to sort
** chunk_len == size of the chunk in bytes for send/recv commands 240
*/
chunk_size = num_items / num_nodes;
chunk_len = chunk_size * sizeof(int);
for(node_idx = 0; node_idx < num_nodes; node_idx++)
{
    csend(PKT_LENGTH_MSG, &chunk_size, sizeof(chunk_size),
          node_idx, PID);
}

/*
** Send each node its part of values[] to sort.
*/
for( node_idx = 0; node_idx < num_nodes; node_idx++)
{
    csend(PKT_DATUM_MSG, values+(chunk_size*node_idx), chunk_len,
          gray(node_idx), PID);
}

/*
** Wait for nodes to send data back to me!
*/
for(i=0;i < num_nodes; i++)
{

```

do_Shell-ABS(ShellHost.c)

```
crecv(PKT_DATUM_MSG, temp, chunk_len);

ring_pos = ginv(infonode());
for(j = 0; j < chunk_size; j++)
{
    values[ring_pos*chunk_size + j] = temp[j];
}
}                                270

/* Start of code added for PRASE routines */

printf("Waiting for data collection to finish.\n");
while((prase_ptr = fopen("prase_end","r")) == NULL);
system("rm prase_end");
fclose(prase_ptr);                  280

/* End of code added for PRASE routines */

/*
** wait for all nodes to end
*/
waitall(ALL_NODES, PID);

return;
}
```

...ABS

```

/*****+
* FUNCTION NAME: void randarray(long*, long, long, long, long, long, long)
* FUNCTION NUMBER: 2 (ShellHost.c)
* VERSION NUMBER: 1.2
* DATE: 15 Sep 1989
* DESCRIPTION :
*   Function creates an unsorted array of integers between
*   1 and elements with no repeats. The array's sortedness is
*   declared as a percentage (from 0 to 100) and relates to the
*   average distance an element is from its sorted position. For example,
*   for a 100-element array with 50% sortedness, most elements will be
*   less than 100*.50=50 position out of place. The seed to the random
*   number generator is also specified. The array is further
*   specified by pattern, cycles, and invert. These six parameters
*   provide a means for reproducing a specific array.
* INPUT PARAMETERS:
*   array - pointer to an array of integers. It is assumed
*           that memory has already been allocated.
*   seed, sortedness, elements,
*   pattern, numberOfCycles, invert - six parameters from which
*           an interesting input array can be created
* OUTPUT PARAMETERS:
*   array - filled with (un)sorted integers.
* FUNCTIONS CALLED:
*   srand() - random number seed function
*   rand() - random number generator
* HISTORY:
*   (5 May 89) Created for AAARF prototype.
*   (20 July 89) Modified extensively by Ed Williams to accomodate
*           cycles, inverse order, and patterns.
*   (10 August 89) Modified slightly to conform to AAARF
*           requirements.
*   (15 Sept 89) - Version 1.0
*   (1 November 90) Copied to Shell Sort Program
*****+
void randarray(array, seed, sortedness, elements,
               pattern, numberOfCycles, invert)
long array[];
long seed, sortedness, elements, pattern, numberOfCycles, invert;
{
    register int i, j, inc, cycle, displacement, temp;
    float pc_displacement, max_displacement, r_fraction;
    int rand(), srand();
    inc = invert ? -1 : 1;
    pc_displacement = (float)(100 - sortedness)/100.0;
    max_displacement = (float)elements * pc_displacement;
    (void)srand((unsigned)seed);
    j = (inc > 0) ? 1 : elements;
    switch(pattern)
    {

```

291

300

310

320

330

340

```

case 0:          /* linear input data generator (original) */
    for (i = 0; i < elements; i++, j += inc)
        array[i] = j;
    break;
case 1:          /* sawtooth input data generator */
    cycle = 0;
    inc *= numberOfCycles;
    for (i = 0; i < elements; i++)
    {
        array[i] = j;
        j += inc;
        if((j > elements) || (j < 1))
        {
            cycle++;
            j = (inc > 0) ? 1 + cycle : elements - cycle;
        }
    }
    break;
case 2:          /* bitonic input data generator */
    inc *= (2 * numberOfCycles);
    for(i = 0; i < elements; i++)
    {
        array[i] = j;
        j += inc;
        if((j > elements) || (j < 1))
        {
            inc = -inc;
            j = j + 1 + inc;
            if(j > elements) j += inc;
        }
    }
    break;
}

/****** APPLY SORTEDNESS FACTOR *****/ 380

for ( i = 0 ; i < elements ; i++ )
{
    do
    {
        r_fraction = (float)rand() / (float)MAX RAND;
        r_fraction = 2.0 * r_fraction - 1.0;
        displacement = (int)( ( max_displacement + 1.0 ) * r_fraction);
    }
    while (ABS(displacement) >= MAX( (elements-i), i ) );

    j = i + displacement; 390
    if ( ( j >= elements ) || ( j < 0 ) )
        j = i - displacement;

    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

```

ABS-MIN(ShellHost.c)

 }

...MIN

400

Appendix B. PRASE Data Formats

This appendix describes the data formats used to transfer data between the instrumented program on the remote system and the display system, and the utility programs developed to display and manipulate the data outside the AAARF environment.

Trace Data Format

The trace data records use the same format as the PRASE [19] system. This compatibility was maintained so that (to a limited extent) trace data from original PRASE instrumentation can still be displayed. The data format for the original PRASE implementation is contained in Appendix C of the PRASE User's Manual [20], but it is covered here as well because the process of porting the system to the iPSC/2 changed the lengths of the individual items. The relevance of the data is also discussed in the context of the parallel animation system. Figure B.1 shows a pictorial representation of a trace record.

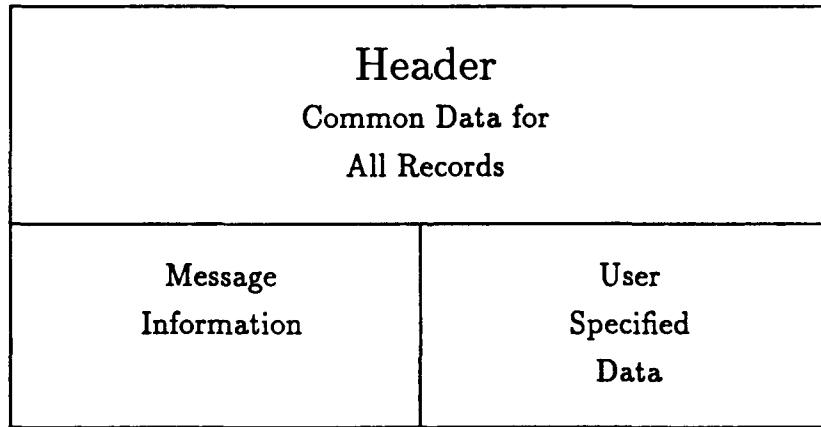


Figure B.1. PRASE trace data record

The record is constructed using a C structure containing a union, so that the first part of the record can remain fixed for all records, while the last part can vary to accommodate different records. The C type declaration for the trace record is shown in Figure B.2. Each member of the structure is explained in detail below. All time fields are

in microseconds since the start of the trace run. The reference time is taken after the node processes are synchronized, so time zero is close to the same physical time on all nodes.

```
*****  
* This is the structure that defines the  
* format of the data coming from the  
* monitoring routines. See the documentation  
* for a description of the contents.  
* NOTE: The int/long/unsigned data in the  
* record comes from the hypercube in  
* byte-reversed order (thanks to Intel).  
*****  
typedef struct  
{  
    char        record_type[6];  
    long        recording_node;  
    long        recording_pid;  
    unsigned long begin_time;  
    unsigned long end_time;  
    union  
    {  
        struct  
        {  
            long channel_num;  
            long message_type;  
            long message_size;  
            long message_count;  
            long addressed_node;  
            long addressed_pid;  
            long position_marker;  
        } message;  
        char    char_marker[14];  
        double double_marker;  
        float  float_marker;  
        int    int_marker;  
        long   long_marker;  
        short  short_marker;  
    } data;  
} PRASE_RECORD;
```

Figure B.2. PRASE Structure Definition

record_type This is a six-character field that identifies the type of the record. It is used to determine which member of the union to use to access the rest of the record. The valid entries in this member are:

- **init** — this record marks the end of the initialization of the instrumentation routines. This record is used by the other parts of the system to determine the number of processes active for tracing.

- **end** — this record marks the end of the tracing on the node processors.
- **flick** — corresponds to a **flick()** system call.
- **led** — corresponds to a **led()** system call.
- **iprobe** — corresponds to a **iprobe()** system call.
- **recv** — corresponds to a **irecv()** system call.
- **isend** — corresponds to a **isend()** system call.
- **crecv** — corresponds to a **crecv()** system call.
- **crecvb** — this is a new record type that indicates a node process has entered a **crecv()** call.
- **csend** — corresponds to a **csend()** system call.
- **mchar** — allows the user to insert a 14 character message into the trace data stream.
- **mdbl** — allows the user to insert a C double value into the trace data stream.
- **mfloat** — allows the user to insert a C float value into the trace data stream.
- **mint** — allows the user to insert a C int value into the trace data stream. This is a 32-bit quantity stored in low-byte-first order.
- **mlong** — allows the user to insert a C long value into the trace data stream. This is a 32-bit quantity stored in low-byte-first order.
- **mshort** — allows the user to insert a C short value into the trace data stream. This is a 16-bit quantity stored in low-byte-first order.

The **open**, **close**, **grled**, **rdled**, **stat**, and **slog** record types were removed from the iPSC/2 version because they have no equivalents in the new system library. The **dump** record type is not used in the animation system to reduce overhead.

recording_node This is an integer (32-bit) that contains the node number of the processor that created this trace record.

recording_pid This is an integer (32-bit) that contains the process ID of the process that created this trace record.

begin_time This is the relative time at the start of the operation that is being traced.

end_time This is the relative time at the end of the operation that is being traced, but before the trace record is written.

data This is the union that maps the rest of the record differently depending on the type.

If the type is one of the user-provided record types, the appropriate **_marker** member is used. If the type is one of the message-passing calls, the **message** member is used. Otherwise, this member is unnecessary and should be ignored. The **message** structure is described below:

channel_num This is an integer (32-bit) that is unused and is always 99.

message_type This is an integer (32-bit) that contains the user-specified type of the message.

message_size This is an integer (32-bit) that contains the length of the message in bytes.

message_count This is an integer (32-bit) that is unused and is always -1.

addressed_node This is an integer (32-bit) that contains the processor number of the originator of a received message or the destination of a sent message.

addressed_pid This is an integer (32-bit) that contains the process ID of the originator of a received message or the destination of a sent message.

position_marker This is an integer (32-bit) that is unused and is always -1.

The **channel_num**, **message_count**, and **position_marker** members are maintained for compatibility with PRASE — they are not used in this system.

Algorithm Data Format

The format of the data in the algorithm data stream can be discussed at two different levels. At a low level, it is simply a variable-sized data block that contains a header that describes the source and length of the block (see Figure B.3). The fixed-length header is processed first, then the data follows.

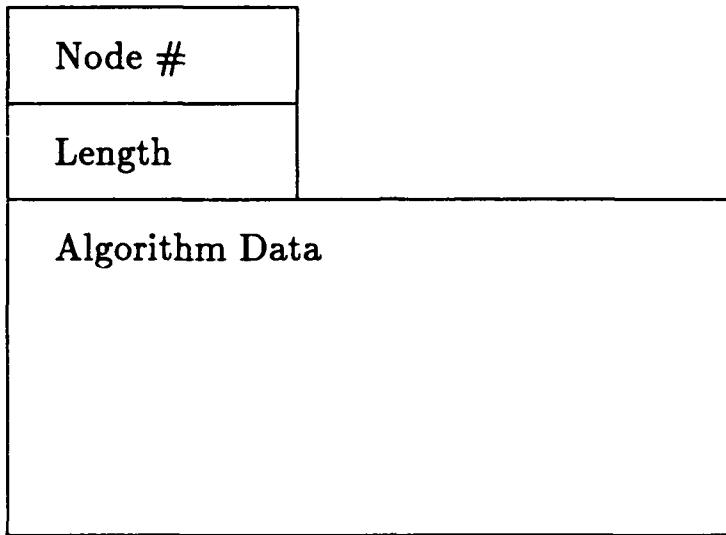


Figure B.3. Algorithm data record

The header consists of two 32-bit integers in low-byte-first order. The first integer is the number of the node processor that originated the block. The second integer is the length of the block in bytes. The two integers and the data block are put into the data stream in three separate `write()` calls, and therefore must be read from the stream in the same fashion. When reading from a file, this restriction does not apply.

From a higher point of view, the format of the data block can be anything that is needed. There is only one restriction on the data block — it must fit the communications medium. The only link that has a size restriction is the link from the nodes to the host; the size limit for a message is 256k bytes. Within this limit, the data block can contain any type of data that the instrumentation requires. Each class of algorithms will usually have different algorithm data formats; but a single algorithm class can also use several different formats.

Appendix C. *AAARF Programmer's Manual*

This appendix contains the AAARF Programmer's Manual. The changes made to AAARF are major enough to cause significant changes in this manual. The entire manual is contained in this appendix, even though only portions were modified. The original manual was written by Keith Fife [10].

Since the manual is intended to be a separate document, its page numbers do not follow the numbering system for this thesis.

AAARF Programmer's Guide

Keith Carson Fife
Captain, USAF

Edward Michael Williams
Captain, USAF

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433

December, 1990

Contents

1	Introduction	6
2	Overview	8
2.1	Algorithm Animation Concepts and Definitions	8
2.2	AAARF Architecture	11
2.3	Windows	13
2.4	AAARF Directory Structure	15
2.5	SunView	17
2.6	Program Documentation	17
2.7	Client-Programmer Tasks	20
3	The AAARF Main Process	21
3.1	aaarf.c	22
3.2	aaarfMenu.c	23
3.3	aaarfControl.c	24
3.4	aaarfEnvironment.c	25
3.5	aaarfWindows.c	26
4	Class-Common Library	28
4.1	aaarfCommon.c	35
4.2	aaarfMaster.c	37
4.3	aaarfRecorder.c	38
4.4	aaarfViews.c	40
4.5	aaarfUtilities.c	42
5	Class-Specific Functions	43
5.1	Creating a Working Directory	44
5.2	Testing a New Algorithm Class	44
5.3	General Requirements	45
5.4	Class-Specific Data	47
5.5	Input Functions	48
5.6	Algorithm Functions	49
5.7	View Functions	50
5.8	Status Functions	51
5.9	Control Functions	52
6	Parallel Views Library	53
6.1	PControl.c	55
6.2	PViews.c	56

6.3	PVXXXX.h (Draw and Update View)	59
6.4	Utils.c	60
6.5	PRASEBG.c	61
6.6	PRASE.h	62
7	Parallel Program Instrumentation	64
7.1	Instrumentation Functions	64
7.2	aaarf_clct	66
7.3	server	66
8	Animating an Algorithm	67
8.1	Animation Process	67
8.1.1	Analyzing the Algorithm	67
8.1.2	Displaying the Algorithm	68
8.1.3	Instrumenting the Program	69
8.1.4	Iteration	73
9	AAARF Projects	74

List of Figures

1	Algorithm Animation Components	9
2	Algorithm Animation System	10
3	AAARF Levels of Execution	11
4	AAARF Windows	13
5	Multiple Algorithm Windows	14
6	Multiple Views within an Algorithm Window	15
7	AAARF Directory Structure	16
8	SunView Notifier Model [17:23]	18
9	Creating a Working Directory for the Bin Packing Class	44
10	Background Process Structure	46
11	Performance Views Control Panel	58
12	Typical instrumentation header file	70
13	Instrumenting the <code>main</code> node function	71
14	Instrumenting the host process	72

List of Tables

1	Module Header Information	19
2	Function Header Information	19
3	Function Prototypes for <i>aaarf.c</i>	22
4	Function Prototypes for <i>aaarfMenu.c</i>	23
5	Function Prototypes for <i>aaarfControl.c</i>	24
6	Function prototypes for <i>aaarfEnvironment.c</i>	25
7	Function Prototypes for <i>aaarfWindows.c</i>	27
8	<i>aaarfWindows.c</i> Data Structures	27
9	<i>commonDefines.h</i> (1 of 2)	30
10	<i>commonDefines.h</i> (2 of 2)	31
11	<i>aaarfDefines.h</i>	32
12	<i>aaarfIPC.h</i>	33
13	<i>classCommon.h</i>	34
14	Function Prototypes for <i>aaarfCommon.c</i>	36
15	Function Prototypes for <i>aaarfMaster.c</i>	37
16	Function Prototypes for <i>aaarfRecorder.c</i>	39
17	<i>aaarfRecorder.c</i> Data Structures	39
18	Function Prototypes for <i>aaarfViews.c</i>	41
19	Function Prototypes for <i>aaarfUtilities.c</i>	42
20	Required Input Functions	48
21	Required Algorithm Functions	49
22	Required View Functions	50
23	Required Status Functions	51
24	Required Control Functions	52
25	Function Prototypes for <i>PControl.c</i>	55
26	Function Prototypes for <i>PViews.c</i>	56
27	Function Prototypes for a typical <i>PVXXXX.c</i>	59
28	Function Prototypes for <i>Utils.c</i>	60
29	Trace Data Structure	63

Revisions

December 1989 This document was first released by Capt Keith Fife as a part of his Master's Thesis[4].

December 1990 Material was added to document the enhancements made by Capt Ed Williams to support research for his Master's Thesis[18]. The major changes are:

- The class-common library was modified to be a separately compiled unit that is linked into the class-specific program
- The parallel views library is documented in Section 6
- Section 8 contains a description of the process of animating an algorithm.

AAARF Programmer's Guide

Captain Keith C. Fife

Captain Edward M. Williams

1 Introduction

The AFIT Algorithm Animation Research Facility (AAARF) is an interactive algorithm animation system. It provides a means for visualizing the execution of algorithms and their associated data structures. AAARF allows the user to select the type of algorithm, the input to the algorithm, and the views of the algorithm. Several control mechanisms are provided, including stop, go, reset, variable speed, single-step, and break-points. Other features of AAARF include:

- Multiple Algorithm Windows
- Simultaneous Control of Multiple Animations
- Animation Environment Save and Restore Capability
- Multiple View Windows within each Algorithm Window
- Animation Record and Playback
- Algorithm State Display and Interrogation Capability
- Master Control Panel for monitoring and modifying the input, algorithm, view, and control parameters to an algorithm animation.

AAARF runs on Sun3TM and Sun4TM workstations using SunOSTM (Sun Microsystem's version of the AT&T UNIXTM operating system) and the SunViewTM window-based environment. AAARF is designed for use with color monitors. It can be used with monochrome monitors, but the displays are not as informative. AAARF is written in the C programming language.

AAARF has two types of users: *end-users* who view and interact with the algorithm animations, and *client-programmers* who develop and maintain the AAARF program and its algorithm animations. This manual is intended for *client-programmers*. It describes the overall AAARF implementation and provides a guide for creating new algorithm animations for AAARF. End-users should refer to the *AAARF User's Manual* [3].

Client-programmers should have experience with AAARF as end-users and be familiar with the terms and concepts associated with algorithm animation. Experience with C, UNIX, and SunView is assumed. The following references are recommended:

- *Getting Started with SunOS: Beginner's Guide* [12]
- *Setting Up Your SunOS Environment: Beginner's Guide* [15]
- *The SunView1 Beginner's Guide* [16]
- *The C Programming Language* [8]
- *C Programmer's Guide* [11]
- *Network Programming* [13]
- *SunView1 Programmer's Guide* [17]
- *Pixrect Reference Guide* [14]
- *Algorithm Animation* [2]
- *The Graphical Representation of Algorithmic Processes* [4]

The next section presents an overview of AAARF and an introduction to terms and concepts associated with algorithm animation; this is the starting point for new AAARF programmers. After gaining some familiarity with AAARF, this section may be used only as a reference. Section 3 discusses the AAARF main process. This section is intended primarily as a maintenance guide for the AAARF main process; however, it is also recommended reading for programmers who are creating new algorithm animations.

Section 4 introduces the animation level of AAARF. Aspects of AAARF algorithm animations common to all algorithm classes are presented here. Section 5 discusses the class-specific aspects of creating algorithm animations. Sections 4 and 5 are essential references for the development of new algorithm animations. A library for displaying parallel performance data is described in Section 6 followed by a description of the instrumentation routines for gathering data from parallel processes in Section 7.

The process of animating an algorithm is presented in Section 8. Section 9 presents some ideas and suggestions for AAARF extensions and programming projects.

2 Overview

This section presents an introduction to several topics of interest to AAARF programmers and animation developers. It outlines some basic concepts of algorithm animation and presents an overview of the AAARF system architecture. The Sun-View Notifier and program documentation are also discussed. It is important to understand the concepts presented in this section before beginning to develop AAARF applications. See the references listed in Section 1 for more detailed presentations.

2.1 Algorithm Animation Concepts and Definitions

These concepts, extracted from [4] and [2], are critical for maintaining AAARF and its algorithm animations.

Animation Components An algorithm animation consists of three components: an input generator, an algorithm, and one or more animation views (see Figure 1).

Input Generator An input generator is a procedure which provides input to an algorithm; the input may be generated randomly, read from a file, or entered by the user.

Animation View Animation views are graphical representations of an algorithm's execution. The view refers to both the graphical representation and the software that generates the view.

Interesting Event (IE) Animation views are driven by interesting events that occur during the execution of an algorithm. Interesting events are input events, output events, and state changes that an algorithm undergoes during its execution. The type, quantity, and sequence of IEs for a particular algorithm distinguish it from other algorithms.

Algorithm Class Algorithms which operate on identical data structures and perform identical functions are from the same algorithm class. The *Array Sort* class includes *quick sort*, *heap sort*, *bubble sort*, and other in-place sort algorithms. Algorithms from the same class generally share input generators and views, although certain views and input generators may be ineffective with particular algorithms. For instance, the *tree* view is very meaningful for heap sort, but nearly useless for any other sort.

Algorithm Animation System Model The elements of an algorithm animation system are depicted in Figure 2 and include the environment manager, component library, and library manager.

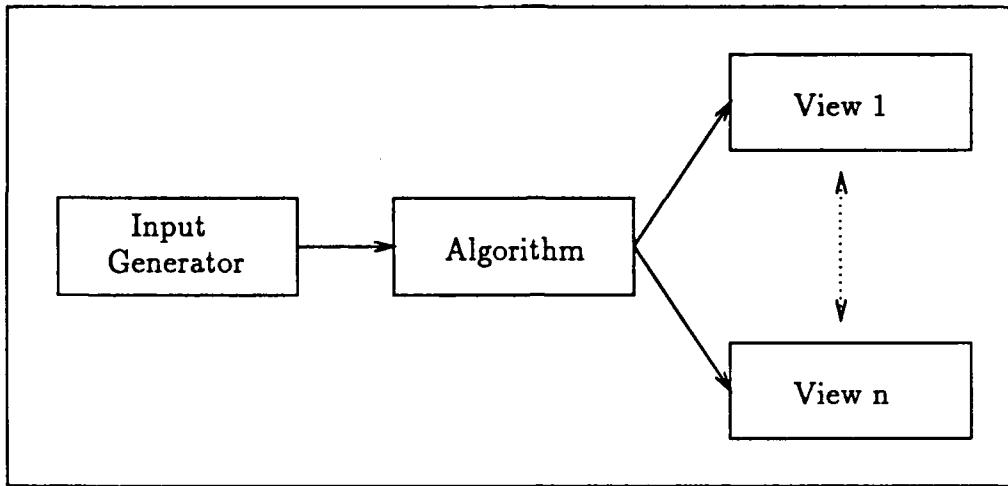


Figure 1: Algorithm Animation Components

Environment Manager The environment manager is the process with which end-users interact to select, control, and view algorithm animations. The environment manager supports multiple algorithm animations and provides a means for controlling their execution.

Component Library The component library is a collection of algorithm components arranged by algorithm class. The environment manager calls routines from the component library.

Library Manager The library manager is the process with which client-programmers interact to maintain the component library. The library manager provides a means to add and delete classes and to create, modify, or delete algorithms, input generators, and views within algorithm classes. Although the environment manager uses the component library directly, changes made by the library manager should not affect the environment manager. The modified components should be immediately ready for use by the environment manager.

Parameterized Control User-selectable parameters are associated with each component. *Algorithm parameters* affect some aspect of how the algorithm executes. For example, with a quick sort algorithm, what partitioning strategy should be used; as the partitions get smaller, at what point should another type of sort be used; what other type of sort should be used. *Input parameters* affect the input generator – what seed is used to generate a set of random numbers; how

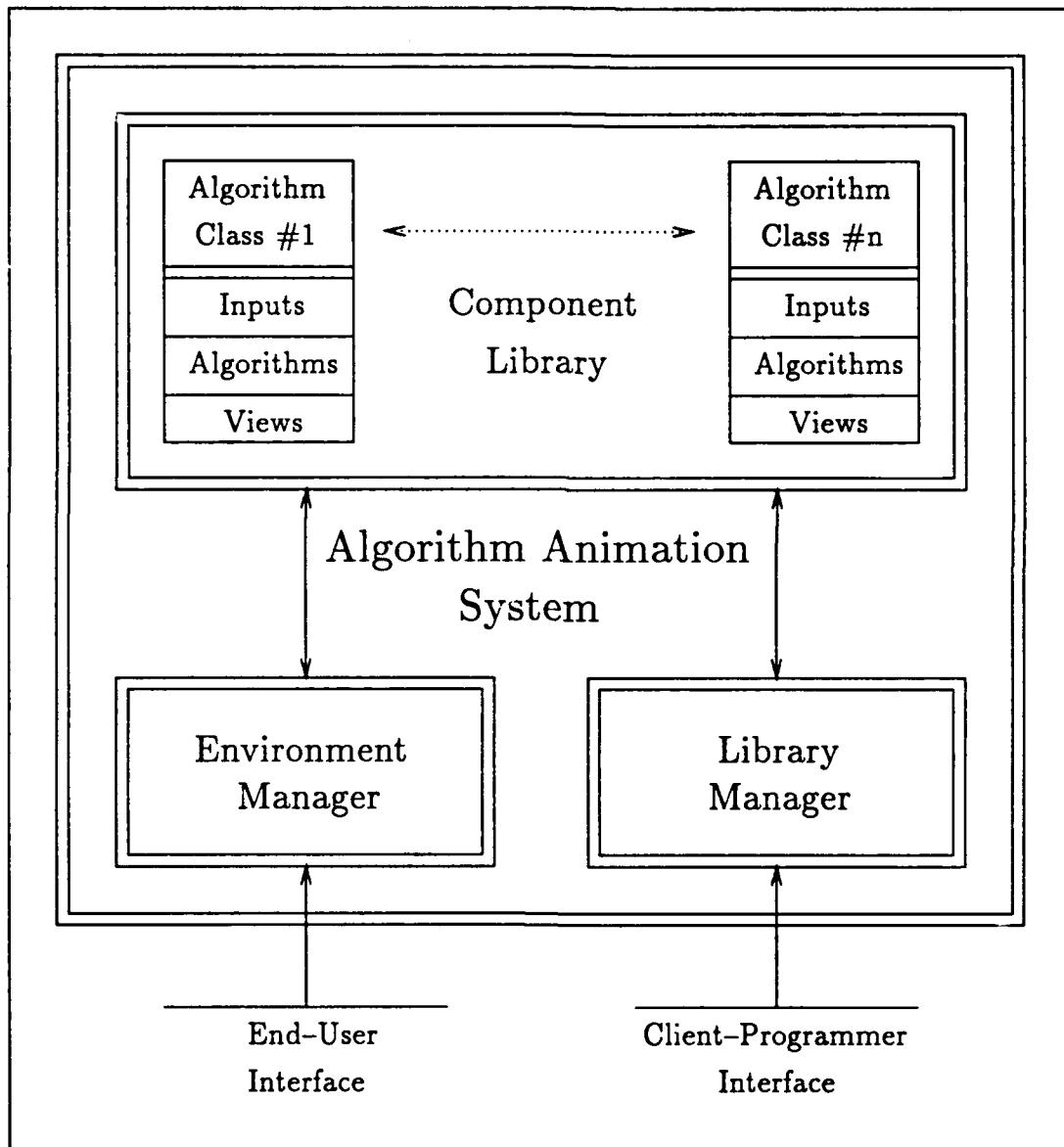


Figure 2: Algorithm Animation System

“sorted” is a set of unsorted numbers; what is the general form of a series of numbers. *View parameters* affect how the animation is displayed in the view window. For example, what shape is associated with the nodes in a graph; how should an arbitrary graph be positioned.

2.2 AAARF Architecture

AAARF uses three levels of execution linked via UNIX sockets [13:191-217] to implement the algorithm animation system model. Sockets are the interprocess communication mechanism provided by the UNIX operating system. Figure 3 shows the levels of execution.

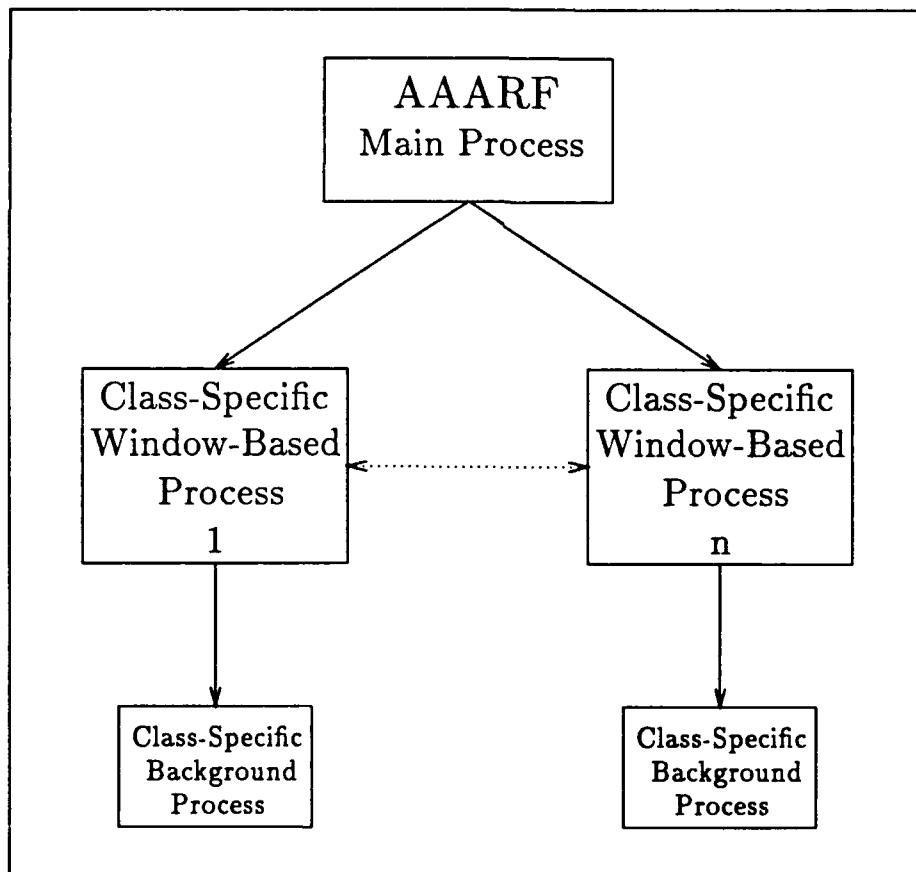


Figure 3: AAARF Levels of Execution

The AAARF Main Process

AAARF's top-level process acts as the environment manager. It provides the main screen, a mechanism for saving and restoring animation environments, a mechanism for controlling multiple algorithm animations simultaneously, and a means for starting new algorithm animations.

- *In general, the AAARF main process requires the least attention from client-programmers.*

The Class-Specific Window-Based Process

The second level of execution is the class-specific window-based process. The window-based level of execution provides the animation views, an animation recorder, a status display for interrogating the state of the algorithm, and a master control panel for monitoring and modifying the parameters that affect the animation. The view component of Figure 2 is implemented at this level.

The Class-Specific Background Process

The third level of execution is transparent to the user; this level is the implementation of the input generator and algorithm components of the algorithm animation system model (Figure 2). It provides the window-based level with the IEs that drive the animation. The IEs can come from an implementation of the algorithm within the background process or from a program on another system communicating with the background process.

2.3 Windows

AAARF uses three types of windows to provide the user with multiple simultaneous algorithm animations and multiple views of each animation. Figure 4 shows the types of windows used by AAARF and their relationship to one another.

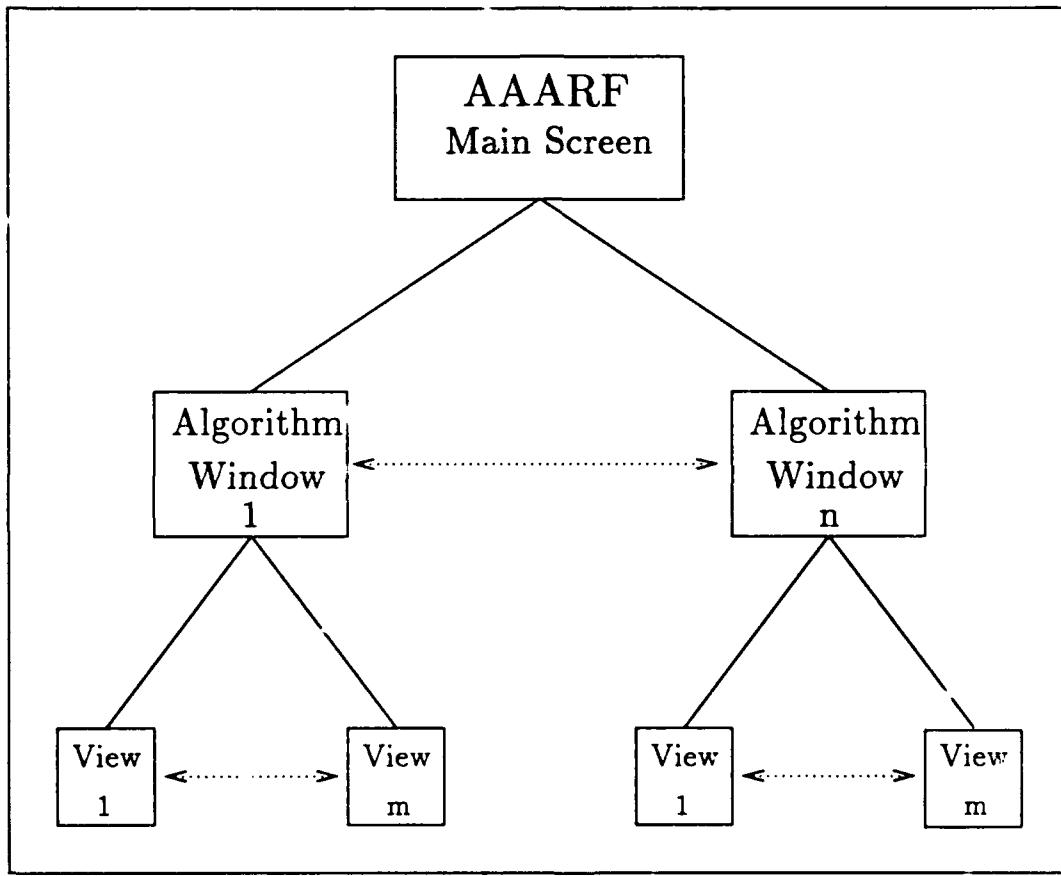


Figure 4: AAARF Windows

AAARF Main Screen

This is a full-screen window within which all interaction with AAARF is contained. The main screen currently supports up to four algorithm windows.

- *The actual number of algorithm windows possible is dependent on the system configuration, cpu load, and memory availability.*

Algorithm Window

All animations take place within algorithm windows. Every algorithm window is associated with a particular algorithm class. Algorithm windows may be created and destroyed at any time, but no more than four can be active at any time. Each algorithm window supports an animation recorder, a master control panel, a status display, and up to four view windows. Algorithm windows can be resized and moved anywhere on the AAARF screen; they may overlap one another.

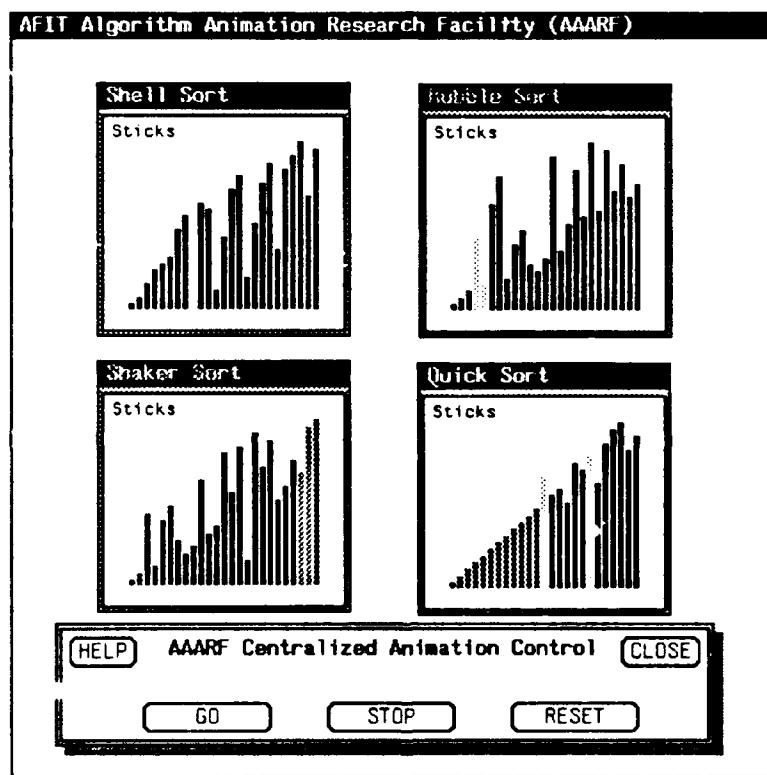


Figure 5: Multiple Algorithm Windows

View Window

View windows, or views, are windows associated with a particular view of an algorithm. Every algorithm window has at least one and as many as four active view windows. View windows can be resized and moved, but they cannot grow outside of the algorithm window and they may not overlap.

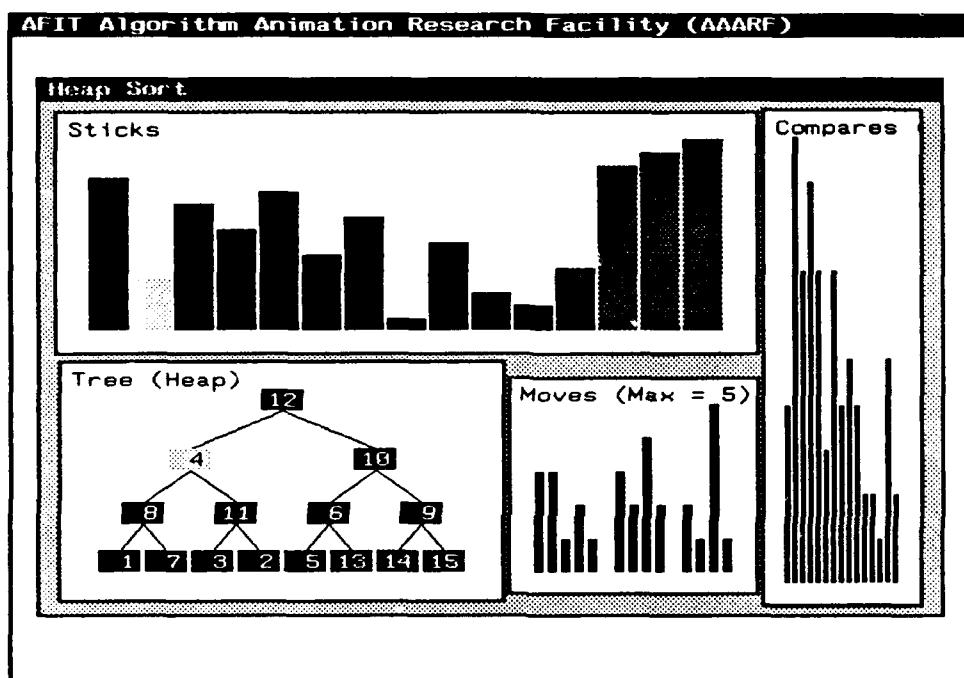


Figure 6: Multiple Views within an Algorithm Window

2.4 AAARF Directory Structure

The AAARF directory structure consists of the *aaarf* parent directory and four subdirectories: *bin*, *main*, *common*, and *include*. Any number of class subdirectories may also be included, but there is no requirement for them to exist within the *aaarf* directory.

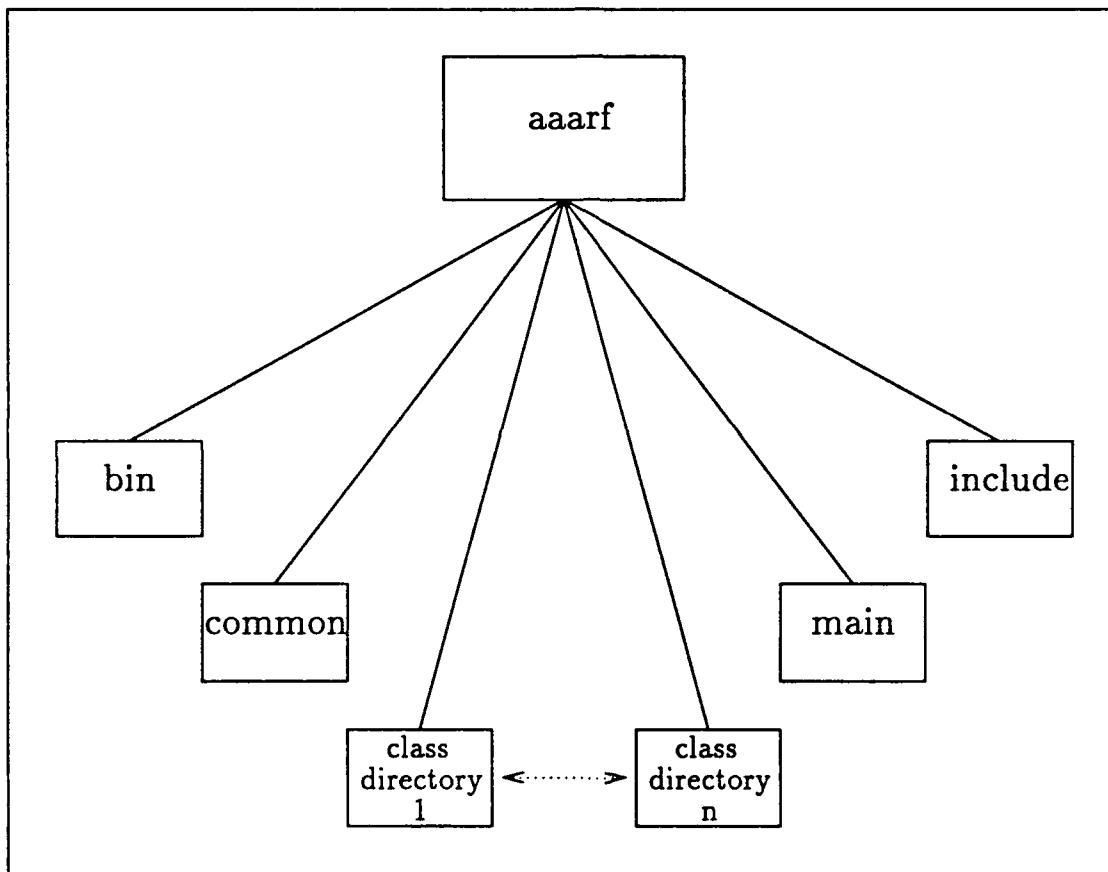


Figure 7: AAARF Directory Structure

The *bin* subdirectory contains the *aaarf* executable image, the class-common library, and the default AAARF class file (*.aaarfClasses*). The AAARF classes file lists the available algorithm classes and the name of the executable file for each class. Section 3.2 addresses the AAARF class file in more detail. Typically, algorithm class executables are also stored in *bin*, but it is not a requirement.

The *main* subdirectory contains source code for the main AAARF process. Section 3 discusses the files in this subdirectory.

The *common* subdirectory contains source code common to all algorithm classes. Client-programmers use this source code as a starting point for building new algorithm animations. Section 4 discusses the files in this subdirectory.

The *include* subdirectory contains the header files and icons used by AAARF.

If class subdirectories exist, they contain the source code for their respective algorithm class executables. Section 5 discusses the functions required for creating an algorithm animation.

2.5 SunView

SunView is an object-oriented system that provides a set of visual building blocks for assembling user interfaces. SunView objects include windows, pointers, icons, menus, alerts, panel items, and scrollbars. AAARF makes extensive use of SunView objects; only the background process does not use SunView.

SunView is a *notification-based* system. The Notifier acts as the controlling entity within an application, reading UNIX input from the kernel, and formatting it into higher-level *events*, which it distributes to the appropriate SunView objects. The Notifier *notifies*, or calls, various procedures which the application has previously registered with the Notifier. These procedures are called *notify procedures*. The SunView Notifier model is shown in Figure 8.

2.6 Program Documentation

The existing source code is fully documented in accordance with AFIT System Development Documentation Guidelines and Standards [5]. Most functions are less than one page long and most modules contain less than ten functions. The existing AAARF source code should be used along with this manual in developing new animations.

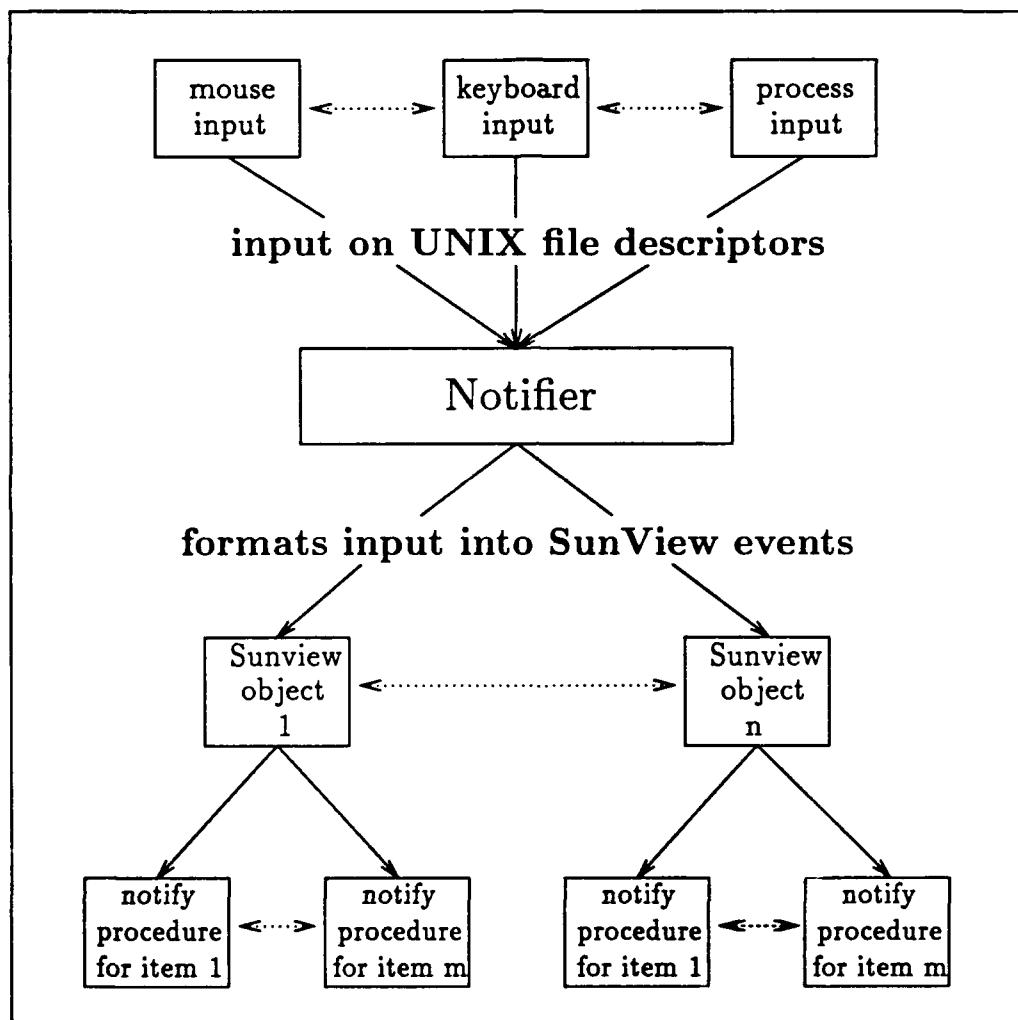


Figure 8: SunView Notifier Model [17:23]

- **FILE** : The module name (UNIX file name)
- **PROJECT** : AFIT Algorithm Animation Research Facility (AAARF)
- **DATE** : Date of current version number
- **VERSION** : Current version number
- **AUTHOR** : Person or persons who are responsible for the module.
- **DESCRIPTION** : Description of the module's function.
- **FUNCTIONS** : List of functions implemented in the module.
- **REFERENCES** : References which could help explain what's happening in the module.
- **HISTORY** : List of major changes to the module

Table 1: Module Header Information

- **FUNCTION NAME** : Name of the function.
- **FUNCTION NUMBER** : Function's number within this module.
- **VERSION NUMBER** : Current version number.
- **DATE** : Date of current version number.
- **DESCRIPTION** : Detailed description of what the function does.
- **INPUT PARAMETERS** : List and description of input parameters.
- **OUTPUT PARAMETERS** : List and description of output parameters.
- **FUNCTIONS CALLED** : List of functions called.
- **HISTORY** : List of major changes to the function.

Table 2: Function Header Information

2.7 Client-Programmer Tasks

Client-programmers are most frequently concerned with the window-based level and its corresponding background process. Undoubtedly, their most difficult task is identifying IEs and developing meaningful views to represent them; this task is described in Section 8. Other typical tasks include:

- Adding new input generators and modifying existing input generators for an existing algorithm class.
- Adding new algorithms and modifying existing algorithms for an existing algorithm class.
- Adding new views and modifying existing views for an existing algorithm class.
- Developing new algorithm classes.
- Improving and extending capabilities of main AAARF process.

3 The AAARF Main Process

The AAARF main process is not associated with any particular algorithm animation, but provides an environment in which *any* algorithm animation can be controlled. The AAARF main process is stand-alone, but without animation processes to run, it is not very interesting. The source code for the AAARF main process is contained in the *main* subdirectory and consists of the following files:

- *aaarf.c* Creates the main AAARF screen and displays the welcome screen.
- *aaarfControl.c* Creates and maintains the Central Animation Controller.
- *aaarfEnvironment.c* Creates and maintains the AAARF Environment Control Panel.
- *aaarfMenu.c* Creates and maintains the AAARF main menu.
- *aaarfWindows.c* Opens and closes algorithm windows. Maintains accounting information regarding open algorithm windows.

Each module has a corresponding header file in which the module's functions, globals, and data structures are declared. The following sections discuss each of the five modules.

3.1 aaarf.c

This is the top level module for the AAARF main process. It creates the AAARF main screen and calls functions from the other modules to create their respective contributions to AAARF. Table 3 lists the functions included in *aaarf.c*.

The *aaarfEventDispatch()* function interposes user input to AAARF such that:

- The cursor's screen position is saved when the right mouse button is depressed. Windows and panels appear at the most recently recorded cursor position.
- If AAARF is iconic, the left mouse button deiconifies AAARF. Otherwise the left mouse button is ignored, thus disabling its usual function of popping and pushing windows.
- If AAARF is iconic, the middle mouse button is used to move the icon. Otherwise, the middle button is ignored, forcing the AAARF main screen to be either full-size or iconic. This eliminates any problems with hidden or lost algorithm windows among other application windows.
- Ignore ACTION keys. Forces use of mouse and eliminates another way to loose windows.

A polling timer is associated with *aaarfShowWelcomeMessage()* in *main()*. When *window_main_loop()* is entered, the Notifier calls *aaarfShowWelcomeMessage()*. It displays the AAARF help screen and disables its polling timer. Occasionally, on a Sun3, the help screen is displayed before the AAARF main screen – the solution has been to set the polling timer for a longer interval.

```
int main(int, char**)
Notify_value aaarfEventDispatch(Frame, Event, Notify_arg, Notify_event_type)
Notify_value aaarfShowWelcomeMessage(Notify_client, int)
void aaarfHelp()
```

Table 3: Function Prototypes for *aaarf.c*

3.2 aaarfMenu.c

This module creates the main menu. In doing so, it registers a `notify_procedure` for each menu item with the Notifier. When a menu item is selected, the Notifier evokes the appropriate function. Table 4 lists the functions implemented in `aaarfMenu.c`.

At startup, `main()` calls `setupMainMenu()` which looks for the `AAARFCLASSES` environment variable. If it's set to a valid AAARF class file, that file is used to generate the algorithm class menu. Otherwise, the default AAARF class file, `.aaarfClasses`, is used.

The AAARF class file contains a list of algorithm class names and their corresponding executable image file names. `setupMainMenu()` reads the class names into the `classStrings` array used to define menu items. It reads the executable filenames into the `classPrograms` array used by `aaarfWindows.c` to execute the animations.

void	setupMainMenu()
caddr_t	menuHelp(Menu, MenuItem)

Table 4: Function Prototypes for `aaarfMenu.c`

3.3 aaarfControl.c

This module creates and manages the central control panel. Table 5 lists the functions implemented in *aaarfControl.c*.

`setupCentralControl()` registers `centralControlDispatch()` with the Notifier. `openControlPanel()` is registered by `setupMainMenu()` in *aaarfMenu.c*.

`centralControlDispatch()` handles most panel events internally, but calls `closeControlPanel()` and `controlHelp()` to close the panel and display a help screen. It uses `aaarfAnnounce()` to send control commands to the active algorithm animations.

```
void setupCentralControl()
void centralControlDispatch(Panel_item, struct inputhead*)
caddr_t openControlPanel(Menu, Menu_item)
void closeControlPanel()
void controlHelp()
```

Table 5: Function Prototypes for *aaarfControl.c*

3.4 aaarfEnvironment.c

This module creates and manages the AAARF environment save and restore feature. The functions implemented in this module are listed in Table 6.

In `setupEnvironmentPanel()`, the initial AAARF environment path is set to the value of the AAARFENV environment variable, if it exists and is a valid directory. Otherwise the initial path is set to the user's current working directory.

`setupEnvironmentPanel()` registers `environmentHelp()`, `buildMenu()`, `environmentDispatch()`, `closeEnvironmentPanel()`, and `getSelection()` with the Notifier. `createMainMenu()` registers `openEnvironmentPanel()`. The file and environment selectors are associated with a menu. `buildMenu()` is associated with each of the menus and `getSelection` is associated with each of the selectors.

Since the paths and files are constantly changing, the path and file menus cannot reliably be built until immediately before they are displayed. When the path or menu selector icons are activated, the `buildMenu()` function is called to create the appropriate menu. First, all the old menu items are destroyed. Then the new menu items are created from the current `pathNames` and `fileNames` lists.

`pathNames` and `fileNames` are built by calls to `getSubdirectories()` and `getDirectory()`. The name lists are built when the environment panel is opened and after every file operation.

The `environmentDispatch()` function handles all the file transactions and error checking.

void	setupEnvironmentPanel()
void	environmentDispatch(Panel.item, struct inputhead*)
caddr_t	openEnvironmentPanel(Menu, Menu.item)
void	closeEnvironmentPanel(Panel.item, struct inputhead*)
Menu	buildMenu(Menu, Menu.generate)
int	getSelection(Panel.item, Event*)
void	environmentHelp(Panel.item, struct inputhead*)

Table 6: Function prototypes for `aaarfEnvironment.c`

3.5 aaarfWindows.c

This module opens, closes, and monitors algorithm windows. Communication with the algorithm processes is via UNIX sockets. The `ALG_WINDOW` and `AAARF_STATE` data structures are used to manage the algorithm windows. Table 7 lists the functions implemented in this module and Table 8 presents the data structure definitions.

`createMainMenu()` registers `openAlgorithmWindow()`, `aaarfIconify()`, and `aaarfKill()` with the Notifier. `aaarfChildMonitor()` is registered as the termination notify_procedure for each algorithm window process.

Both `openAlgorithmWindow()` and `restoreEnvironment()` fork algorithm window processes. They get a socketpair, fork a process, send the process some setup data, and update `aaarfState`.

`aaarfChildMonitor()` is notified each time a child process terminates. It updates the `aaarfState`. This function is *not* called when a child process is killed (UNIX `kill()` function), *provided* the function that kills the child waits (UNIX `wait()` function) for the child process to terminate.

AAARF, all AAARF panels, all open algorithm windows, and all their open panels are iconified and deiconified by `aaarfIconify()`.

`aaarfAnnounce()` provides a channel for sending commands to all algorithm processes. Typical commands are ICONIFY, GO, STOP, and RESET. It always waits for an acknowledgment.

`saveEnvironment()` and `restoreEnvironment()` are the only other functions that communicate directly with the algorithm window process. They also write and read environment parameters to and from disk files.

`killWindow()` is used to kill algorithm window processes. It uses the UNIX `kill()` function. After killing a process it uses the UNIX `wait4()` function to wait for the process to terminate.

```

void setupAlgorithmWindows()
caddr_t openAlgorithmWindow(Menu, MenuItem)
int aaarfAnnounce(int)
Notify_value aaarfChildMonitor(Notify_client, int, union wait*, struct rusage*)
MenuItem aaarfIconify(MenuItem, Menu_generate)
void saveEnvironment(int)
void restoreEnvironment(int)
caddr_t aaarfKill(Menu, MenuItem)
void killWindow(int)

```

Table 7: Function Prototypes for *aaarfWindows.c*

```

typedef struct
{
    int open;
    int pid;
    int socket;
    int class;
}
ALG_WINDOW;

typedef struct
{
    ALG_WINDOW algWindow[MAX_ALG_WINDOWS];
    int numberOpenWindows;
    int CCopen;
    int CCx;
    int CCy;
}
AAARF_STATE;

```

Table 8: *aaarfWindows.c* Data Structures

4 Class-Common Library

Each algorithm class that AAARF calls is actually a stand alone process. Without the communication links to AAARF, the process could be executed without the AAARF main process. Every algorithm class process has the following common features:

- Main function gets setup parameters from AAARF,
- Creates base window for animation,
- Creates the master control panel,
- Creates the algorithm window menu,
- Creates the four animation canvases (views),
- Creates the animation recorder,
- Creates the status display,
- Runs the background process,
- Paints the active canvases,
- Enters the SunView main loop.

The AAARF class-common library presents client-programmers a framework for developing new algorithm animations by providing all the common functions. The client-programmer simply develops the class-specific input, algorithm, view, and control functions. The object code for the functions provided by the class-common library is combined into the library `libAAARF.a` located in the `bin` directory.

Not only does the AAARF class-common library make animation development easier for client-programmers, it also forces a consistent animation interface for the end-user. Every algorithm class supports the same set of tools, and those tools work identically for every algorithm class. After animating one algorithm, end-users can animate any algorithm, regardless of the algorithm class or its client-programmer. The class-common library also ensures a consistent communications interface with the AAARF main process.

The source code for the AAARF class-common library is in the `common` subdirectory and consists of the following files:

- `aaarfCommon.c` Creates the base window for the algorithm class. Creates the algorithm window menu and the animation canvases. It also handles some other miscellaneous initialization tasks.

- *aaarfMaster.c* Creates the master control panel and sets up the control section of the panel.
- *aaarfRecorder.c* Creates the animation recorder. It also provides the primary interface to AAARF.
- *aaarfViews.c* Sets up the view section of the master control panel and creates the status display. Handles the view resizing and repainting and manages the animation.
- *aaarfUtilities.c* This is a set of functions used by the AAARF main process as well as the animation processes.

Each module has a corresponding header file in which the module's functions, globals, and data structures are declared. Each of the five modules are discussed in the following sections.

In addition to the module header files, the following header file defines constants and data structures that are shared between two or more modules:

- *commonDefines.h* Defines constants and data structures shared by all modules within the class-common library. This header file appears as Table 9.

This last set of header files contains definitions that are necessary for both the class-common library and class-specific functions. These header files are contained in the *include* subdirectory:

- *aaarfDefines.h* Defines system-wide constants used by all levels of AAARF. This header file is included in all AAARF source code modules. Table 11 lists the *aaarfDefines.h* header file.
- *aaarfIPC.h* This file defines interprocess communication (IPC) constants and data structures shared by the AAARF main process and the class-common library modules. This header file is listed in Table 12.
- *classCommon.h* This file defines constants and data structures defined by the class-common library and shared with the class-specific modules. Table 13 lists the *classCommon.h* header file.

#define P_VIEWS	0
#define P_LAYOUT	1
#define P_OPEN	2
#define P_WIDTH	0
#define P_HEIGHT	1
#define P_X_POS	2
#define P_Y_POS	3
#define P_COLOR	4
#define P_M_OPEN	5
#define P_M_X_POS	6
#define P_M_Y_POS	7
#define P_R_OPEN	8
#define P_R_X_POS	9
#define P_R_Y_POS	10
#define P_S_OPEN	11
#define P_S_X_POS	12
#define P_S_Y_POS	13
#define P_BREAKPOINTS	0
#define P_SINGLESTEP	1
#define P_SPEED	2
#define AUTO_RESIZE	10
#define USER_RESIZE	20
#define PROP_RESIZE	30
#define LAYOUT_STACKED	0
#define LAYOUT_SIDE_BS	1
#define LAYOUT_CORNERS	2
#define LAYOUT_FILL	3
#define ALG_STOP	0
#define ALG_GO	1

Table 9: *commonDefines.h* (1 of 2)

#define ALG_RESET	2
#define ALG_TOGGLE	3
#define ALG_FINISHED	4
#define ALG_QUERY	5
#define ALG_WIN_OPEN	6
#define ALG_WIN_CLOSE	7
#define REC_OFF	0
#define REC_RECORD	1
#define REC_PLAY	2
#define REC_RESTART	3
#define REC_REVERSE	4
#define REC_FORWARD	5
#define REC_QUERY	6
#define REC_AT_HEAD	7
#define REC_AT_NEXT	8
#define REC_AT_TAIL	9
#define REC_FINISHED	10
#define REC_STOP	11
#define REC_GO	12
#define T_OFF	0
#define T_ON	1
#define T_STEP	2
#define T_QUERY	3
#define TIMER_NULL	((struct itimerval *)0)
typedef struct	
{	
PARAMS input;	
PARAMS algorithm;	
PARAMS view;	
PARAMS control;	
PARAMS window;	
}	
RESTORE_PAK;	

Table 10: *commandDefines.h* (2 of 2)

#define MAX_CLASSES	50
#define MAX_ALG_WINDOWS	4
#define MAX_VIEWS	4
#define MAX_AAARF_NAME_SIZE	15
#define MAX_FILES	100
#define MAX_FILENAME_SIZE	256
#define MAX_PATHS	100
#define MAX_PATHNAME_SIZE	1024
#define DISPLAYED_PATH_LENGTH	25
#define MAX_COMMAND_LINE_SIZE	1124
#define AAARF_MINIMUM_WIDTH	500
#define AAARF_MINIMUM_HEIGHT	500
#define MAX_STRING_LENGTH	1024
#define ASCII_INT_LENGTH	20
#define MIN_WINDOW_WIDTH	150
#define MIN_WINDOW_HEIGHT	150
#define OFF	0
#define ON	1
#define FALSE	0
#define TRUE	1
#define STOP	0
#define GO	1
#define CHILD	0
#define PARENT	1
#define DOWN	0
#define UP	1
#define DEFAULT_SLIDER_WIDTH	256
#define MAX RAND	2147483647
#define MAX(a, b)	((a > b)?a : b)
#define MIN(a, b)	((a < b)?a : b)
#define ABS(a)	((a > 0)?a : a * -1)

Table 11: *aaarfDefines.h*

```

#define NEW_WINDOW           -1
#define ICONIFY              100
#define DEICONIFY             101
#define SIMULTANEOUS_GO      200
#define SIMULTANEOUS_STOP    201
#define SIMULTANEOUS_RESET   202
#define ENV_SAVE              300
#define SEND_DATA             301
#define ACKNOWLEDGE          400

typedef struct
{
    int width;
    int height;
    int xPos;
    int yPos;
    int color;
}
SETUP_PACKET;

```

Table 12: *aaarfIPC.h*

```

#define ANIM_FINISHED 5
#define ANIM_BREAK 3
#define ANIM_CONTINUE 1
#define ANIM_WAITING 0x80

#define PARAM_SIZE 50
typedef int PARAMS[PARAM_SIZE];

typedef struct
{
    PARAMS input;
    PARAMS algorithm;
}
INIT_PAK;

typedef struct
{
    Canvas canvas;
    Pixwin *pixwin;
    int open;
    int view;
    int width;
    int height;
    int xPos;
    int yPos;
}
VIEW_WINDOW;

typedef struct
{
    VIEW_WINDOW window[MAX_VIEWS];
    int numberOpenViews;
}
VIEW_STATE;

```

Table 13: *classCommon.h*

4.1 aaarfCommon.c

This is the top level module for the class-specific window-based process. It creates the base algorithm window, the algorithm window menu, and the animation canvases on which the views are displayed. Table 14 shows the functions included in this module.

`main()` creates the base algorithm window and registers `frameInterposeDispatch()` to monitor window events. It calls several functions to create the standard set of algorithm window tools. `aaarfCommandDispatch()` is registered to monitor the UNIX socket to the AAARF main process.

`frameInterposeDispatch()` catches events in the base algorithm frame before they are dispatched to the appropriate function. Most events are accepted. If a resize event is detected, the old and new sizes are recorded and the window's current views are repainted with calls to `resizeAllViews()` and `repaintAllViews()`.

`aaarfCommandDispatch()` monitors a socket to AAARF. When commands are received, the appropriate functions are called.

The algorithm window is iconified and deiconified in `setAlgWindowState()`. The state of the animation and the state and position of the algorithm panels are preserved.

`createAnimationCanvas()` creates the algorithm canvases (view windows) and registers `canvasInterposeDispatch()` and `canvasEventDispatch()` to handle events within a canvas. `canvasInterposeDispatch()` catches canvas resize events and redisplays all views. `canvasEventDispatch()` catches middle and left mouse button clicks within the canvases to update `elementOfInterest` and control the animation.

`createAlgMenu()` creates the algorithm window menu and registers `menuDispatch()` with the Notifier. `algWindowHelp()` shows a help screen for the algorithm window menu and `algKill()` destroys the algorithm window.

`setAlgWindowTitle()` is called frequently by several different functions to set the algorithm window title bar to reflect the current algorithm name and recorder status.

`openFonts()` opens an array of fonts for writing text to canvases. `fonts[0]` through `fonts[5]` are empty. The remaining array elements point to fonts whose point size is equal to their array index.

int	main(int, char*)
Notify_value	frameInterposeDispatch(frame, Event, Notify_arg, Notify_event.type)
Notify_value	aaarfCommandDispatch(Notify_client, int)
void	setAlgWindowState(int)
void	createAnimationCanvas(int)
Notify_value	canvasInterposeDispatch(frame, Event, Notify_arg, Notify_event.type)
Notify_value	canvasEventDispatch(Window, Event*, caddr_t)
void	createAlgMenu()
caddr_t	menuDispatch(Menu, MenuItem)
void	setAlgWindowTitle()
void	openFonts()
void	algWindowHelp()
void	algVersion()

Table 14: Function Prototypes for *aaarfCommon.c*

4.2 aaarfMaster.c

This module creates the master control panel. The master control panel is divided into four sections: control, input, algorithm, and view. The class-common library provides the control and view sections with some help from the class-specific `setBreakPointsItems()` and `setViewNames()` functions. The input and algorithm sections are provided by the class-specific `addInputSection()` and `addAlgorithmSection()` functions. The functions included in `aaarfMaster.c` are listed in Table 15.

`createMasterControl()` creates the master control panel. It calls functions to setup the four sections of the panel. It registers the `masterEventDispatch()` with the Notifier to monitor the **Help** and **Close** buttons.

`addControlSection()` sets up the control section of the master control panel. It registers `controlEventDispatch()` with the Notifier.

The `getControlParameters()` and `setControlParameters()` functions are used to save and set control parameters for the animation recorder and environment control functions. They both use the `PARAMS` structure for parameter storage. `PARAMS` is simply an array of integers. The size of the array is set by the client-programmer in the class-specific header file.

```
void createMasterControl()
void masterEventDispatch(Panel.item, struct inputhead*)
    int addControlSection(Panel, int)
void controlEventDispatch(Panel.item, struct inputhead*)
void getControlParameters(PARAMS)
void setControlParameters(PARAMS)
```

Table 15: Function Prototypes for `aaarfMaster.c`

4.3 aaarfRecorder.c

This module implements the animation recorder and manages parameter exchanges with AAARF. Table 16 lists the functions implemented in this module and Table 17 presents the data structures defined for this module. The `SETUP_PACKET` structure defined in `aaarfIPC.h` (see Table 12) is also used.

`createRecorder()` creates the recorder panel and registers `recordDispatch()`, `buildMenu()`, and `getSelection()` with the Notifier. The recorder's path and recording selectors work exactly like the path and environment selectors of the environment control panel (Section 3.4).

Recordings are stored in a linked list structure (see Figure 17). The list is managed with three pointers: `head`, `tail`, and `next`. `playIE()` provides IEs by traversing the linked list. The list is doubly linked, but the current version of AAARF does not support reverse playback. Recordings are created with `saveIE()` which accepts an IE, `malloc()`s a new node, and adds the IE. `restartRecording()` sets the `next` pointer to the head of the list. `freeRecorderMemory()` frees the memory previously allocated for a recording. `readRecording()` and `writeRecording()` are used for reading and writing recordings to disk.

`setRecorderState()` controls the recorder functions and maintains state information regarding the animation recorder.

`readParameters()` and `restoreParameters()` are used to restore the animation state. `getParameters()` and `sendParametersToAAARF()` are used to save the animation state.

```

void createRecorder()
void recordDispatch(Panel_item, struct inputevent*)
Menu buildMenu(Menu, Menu_generate)
    int getSelection(Panel_item, Event*)
    int setRecorderState(int)
void readParameters(SETUP_PACKET, int, int)
void restoreParameters()
void getParameters()
void sendParametersToAAARF(int)
void readRecording(int)
void writeRecording(int)
void restartRecording()
void freeRecorderMemory()
IE_PACKET *playIE()
void saveIE(IE_PACKET*)
void recorderHelp()

```

Table 16: Function Prototypes for *aaarfRecorder.c*

```

typedef struct IEnode IE_NODE;

struct IEnode
{
    IE_NODE *lastIE;
    IE_PACKET IEpacket;
    IE_NODE *nextIE;
}

```

Table 17: *aaarfRecorder.c* Data Structures

4.4 aaarfViews.c

This module sets up the view section of the master control panel, creates the status display panel, and manages the animation views. Table 18 lists the functions included in this module. The `VIEW_STATE` data structure is used to manage the view windows. This structure is defined in `classCommon.h` (Table 13).

`addViewSection()` creates the view section of the master control panel and registers `viewDispatch()` with the Notifier. The `getViewParameters()` and `setViewParameters()` functions are used by the recorder and the environment control to get and set view parameters. `restoreViews()` is used to restore a particular view window configuration after a restore operation.

`createStatusDisplay()` creates the status display and registers `statusDispatch()` with the Notifier. It calls `buildStatusDisplay()` to build the class-specific portion of the status display.

`main()` associates a polling timer with `animateTheAlgorithm()`, which is the heart of the animation. There are two ways this timer can be used: it can signal when the next IE is to be fetched, or it can be used to simulate real time. The normal processing that occurs when the timer expires is to get an IE, process the IE, and update the displays. The class-specific function `processIE()` processes the IE and returns a value that indicates the state of the algorithm:

ANIM_FINISHED the algorithm has finished

ANIM_BREAK the algorithm has reached a breakpoint

ANIM_CONTINUE the algorithm is proceeding

In addition to these values, `processIE()` can shift into simulated-time mode by overlaying the status value with `ANIM_WAITING` — this prevents `animateTheAlgorithm()` from getting another IE until `ANIM_WAITING` is removed from the status value, while `processIE()` is called and the displays are updated.

The polling timer is controlled by `setAlgTimer()`; it is set at the end of `animateTheAlgorithm()` if the algorithm state has changed. The display state is controlled and managed by `setDisplayState()`. It is checked at the start of every pass through `animateTheAlgorithm()`; if the display state is not set to `ALG_GO`, nothing happens in that pass.

`resizeAllViews()` sizes all open views according to an argument from the calling function and the current view parameters.

```
int addViewSection(Panel, int)
int viewDispatch(Panel.item, unsigned int, Event*)
void getViewParameters(PARAMS)
void setViewParameters(PARAMS)
void restoreViews()
void resizeAllViews(int)
int setDisplayState(int)
int setAlgTimer(int)
void createStatusDisplay()
void statusDispatch(Panel.item, struct inputheadline*)
Notify_value animateTheAlgorithm(Notify_client, int)
```

Table 18: Function Prototypes for *aaarfViews.c*

4.5 aaarfUtilities.c

This module consists of functions that are common to both the AAARF main process and the algorithm animation processes. The functions are listed in Table 19.

`getScreenResolution` sets the screen height, width, and depth.

`panelFrameEventDispatch()` is a notify procedure which suppresses the default SunView menu from popping up on panels. It is associated with all panels used in AAARF.

`characterFilter()` is used to filter user input from the keyboard; it only passes numbers and alphabetic characters.

`getDirectory()` and `getSubdirectories()` generate a list of files and paths respectively.

`userWarning()` displays an alert message with the argument provided.

```
void getScreenResolution(int*, int*, int*)
Notify_value panelFrameEventDispatch(Frame, Event*, Notify_arg, Notify_event_type)
Panel_setting characterFilter(Panel_item, Event*)
    int getDirectory(char*, char*, char*)
    int getSubdirectories(char*, char*)
void userWarning(Frame, char*)
```

Table 19: Function Prototypes for *aaarfUtilities.c*

5 Class-Specific Functions

The most difficult part of developing an algorithm animation is determining how to meaningfully represent an algorithm's state. There's no procedures or rules for making such a determination; it's usually best to start with the familiar static representations found in text books. After creating a simple view, it's easier to devise more complex and possibly more interesting views. Determining the nature of the graphical representation is a matter of creativity; this section is more concerned with the mechanics of presenting the view.

The AAARF class-common library provides client-programmers a framework for developing new algorithm animations. This section describes the class-specific requirements of the framework and presents recommendations for using the framework to develop new algorithm animations. The *ArraySort* algorithm class is used as an example throughout this section and a minimum implementation of an algorithm class is provided to use as a foundation. The source code for the *ArraySort* class is included in the *ArraySort* subdirectory of the AAARF distribution, and the template source is provided in the *Skeleton* subdirectory.

Section 5.1 describes how to set up a working directory to begin development of a new algorithm animation. Section 5.2 recommends a method for testing algorithm animations under development. Section 5.3 outlines the general requirements for a new algorithm animation. The remainder of the section describes the requirements in more detail.

- *The program development methods presented in this section are just suggestions. Experienced programmers may prefer variations of these methods or even a completely different approach to program development. However, adherence to the suggested methods insures some degree of uniformity among algorithm class implementations, making it easier for any programmer, regardless of their experience level, to modify or complete any another programmers project.*

5.1 Creating a Working Directory

Begin by creating a working directory. Copy the contents of the *Skeleton* directory to use as a template for the development. Either edit the *Skeleton* files or create new files using the *Skeleton* files as a guide. In either case, existing algorithm classes are probably the best guide for developing new algorithm classes. Check with the system administrator to obtain the location of the AAARF source directories; Figure 9 shows typical commands for setting up a working directory.

```
DALI<1> mkdir BinPack
DALI<2> cd BinPack
DALI<3> cp /usr/local/src/aaarf/Skeleton/* .
```

Figure 9: Creating a Working Directory for the Bin Packing Class

5.2 Testing a New Algorithm Class

Test new animations by creating a local AAARF class file that includes the name and path of the algorithm class under development. Copy the default AAARF class file to a local file, add the new algorithm class to the local file's list of classes, and set the AAARFCLASSES environment variable to the local AAARF class file. Now when AAARF is evoked, it uses the local AAARF class file. This allows programmers to work on animations without interfering with other AAARF users. When the animation is ready for public distribution, add its name and path to the default AAARF classes file.

5.3 General Requirements

There are three categories of requirements for developing an algorithm class:

- The class-common framework requires 20 class-specific functions and 5 global variable declarations. The declarations and functions can be distributed among any number of arbitrarily named modules.
- There must be a `paintXXX()` and an `updateXXX()` function for every view that is implemented.
- There must be a background process that maintains a communications link to the window-based process for reporting IEs generated by the algorithms.

There are two different types of background processes, depending on where the algorithm is executing. If the algorithm is actually implemented within the background process, the process usually consists of a `main()` function to establish communications with the window-based process and to call the input generator and the appropriate algorithm, an IE Dispatcher to send interesting events to the window-based level, and the algorithm functions. Figure 10 shows the general structure of this type of background process.

If the algorithm is on another system, the background process serves as an intermediary to collect information from the remote system and present it to the window-based process. The background process still contains a `main()` function that establishes communications with the window-based process, but instead of directly invoking the algorithm, it establishes communications with the remote system, collects the events from the algorithm, and passes them along. The structure shown in Figure 10 would then be implemented on the remote host.

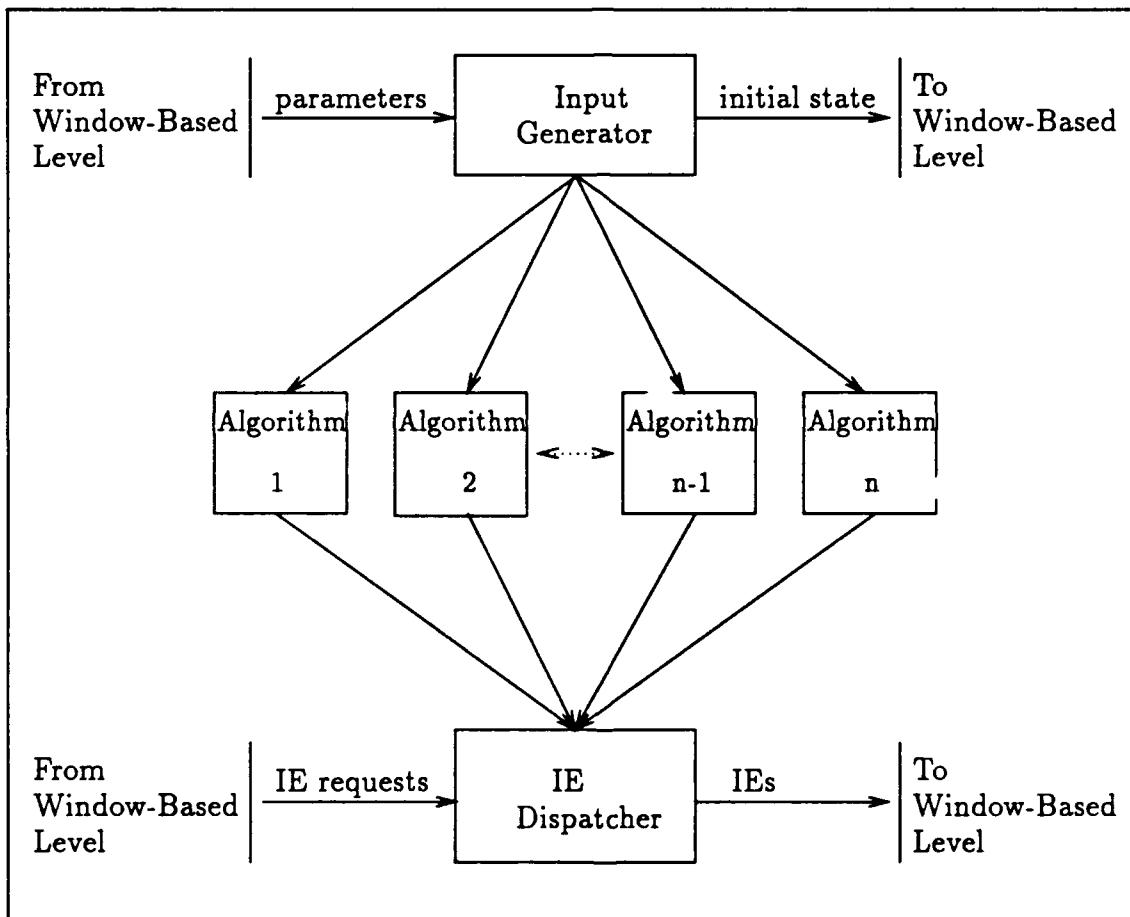


Figure 10: Background Process Structure

5.4 Class-Specific Data

The AAARF class-common functions need five class-specific variable declarations:

- **className** A character array containing the name of the algorithm class.
- **classIcon** A variable containing the icon to be used for the class. *iconedit*, an icon editor available on most Sun workstations, can be used to create an icon, or any icon in the *include* subdirectory can be used. The icon file is used to initialize the variable using SunView macros.
- **TIMER_SCALE** An integer multiplier for the number of milliseconds between calls to *animateTheAlgorithm()*. It may take a few iterations to get this number just right; 3000 is a good starting point. The number must be greater than zero.
- **TIMER_OFFSET** An integer initialized to the minimum number of milliseconds between calls to *animateTheAlgorithm()*. This number must also be greater than zero; one is a good starting point.
- **IE_LENGTH** An unsigned integer initialized to the length of one IE data block.

These declarations are contained in the file *Init.c* in the *Skeleton* subdirectory.

5.5 Input Functions

Three input functions are required by the class-Common library; they are listed in Table 20. `addInputSection()` uses SunView panel items to provide a mechanism for setting parameters to the input generator. The master control panel handle and a row number within the panel are passed to the function. Panel items are positioned beginning at the row number. After all panel items are placed, the row number at which the next panel section can begin is returned. Notify procedures for the input section are not required, but may be used if deemed necessary by the client-programmer.

`getInputParameters()` and `setInputParameters()` provide a means for saving and setting parameters to support the recorder and the environment controller.

These functions are contained in the file `Input.c` in the *Skeleton* subdirectory.

int addInputSection(Panel, int)
void getInputParameters(PARAMS)
void setInputParameters(PARAMS)

Table 20: Required Input Functions

5.6 Algorithm Functions

Four algorithm functions are required for the class-common library; they are listed in Table 21. One function is for adding algorithm panel items to the master control panel, and two functions are used to save and restore the algorithm parameters. The fourth function, `getAlgorithmName` is used to post the algorithm name on the algorithm frame title bar. It returns a pointer to the currently selected algorithm name. A notify procedure for the algorithm panel is not required, but can be used if deemed necessary by the client-programmer.

These functions are contained in the file `Alg.c` in the *Skeleton* subdirectory.

```
int addAlgorithmSection(Panel, int)
char *getAlgorithmName()
void getAlgorithmParameters(PARAMS)
void setAlgorithmParameters(PARAMS)
```

Table 21: Required Algorithm Functions

5.7 View Functions

The class-common library only requires three view functions, but for every view supported by an algorithm class, there must also be a function to paint and update the view within an arbitrarily sized view window. Table 22 lists the required view functions. When one view requires painting or updating, they all do; thus the two functions `paintAllViews()` and `updateAllViews()`. Since the update operation depends on the current IE, `updateAllViews()` requires an IE as an input argument.

`processIE()` accepts the current break point setting and the current IE and returns the animation state: stop, continue, or finished. It also returns a value indicating the way in which the timer is used — see Section 4.4. Within the routine, the animation data structure is modified in accordance with the current IE.

These functions are contained in the file `View.c` in the *Skeleton* subdirectory.

```
int processIE(int, IE_PACKET*)
void paintAllViews()
void updateAllViews(IE_PACKET*)
```

Table 22: Required View Functions

5.8 Status Functions

Four status functions are required; they are listed in Table 23. The status functions share several global variables with the view functions, so they are typically grouped together in the same module. *ArraySortView.c* is an examples of a module that combines the view and status functions.

`buildStatusDisplay()` creates the class-specific entries in the status display. The status display handle and a row number are input parameters. `statusHelp()` provides a help screen for the status display.

`updateElementOfInterest()` is closely related to the view functions in that it must determine which element or aspect of an animation a user has selected.

`updateStatus()` is also related to the view functions – `processIE()` updates the appropriate variables for the status display. The updates are applied to the status panel when this function is called.

These functions are contained in the file `View.c` in the *Skeleton* subdirectory.

void buildStatusDisplay(Panel, int)
void updateElementOfInterest(int, int, Window)
void updateStatus()
void statusHelp(Frame)

Table 23: Required Status Functions

5.9 Control Functions

There are five control functions required by the class-common library. They are listed in Table 24. The `setViewItem()` and `setBreakPointItem()` functions are used to set the class-specific portions of the view section and control section of the master control panel. `masterHelp()` provides the class-specific help screen for the master control panel.

`runAlgorithmInBackground()` is an important function that manages the algorithm state structure and the class-specific background process. It is complemented by the function `termAlgProcessing()`, which terminates the background process in preparation for terminating the run.

`getIE()` is the communications link to the background process. It sends an `IRequest` and receives the next `IE`.

These functions are contained in the file `Control.c` in the *Skeleton* subdirectory.

void	<code>runAlgorithmInBackground()</code>
void	<code>termAlgProcessing()</code>
IE_PACKET	<code>*getIE()</code>
void	<code>setBreakPointItem(PanelItem)</code>
void	<code>setViewItem(PanelItem)</code>
void	<code>masterHelp()</code>

Table 24: Required Control Functions

6 Parallel Views Library

The parallel views library is a collection of functions that are common to most, if not all, animations of parallel algorithms. The library provides two main services:

- A communications link with the remote system hosting the algorithm
- A collection of views that display parallel performance data

The library is implemented as a set of functions that are called by the class-specific functions described in Section 5. There is also a background process that accompanies the library. An algorithm class that constitutes a minimum use of the library is in the *Performance* subdirectory. It is simply the files from the *Skeleton* directory with the calls to the parallel views library inserted in the appropriate places.

This section describes the library in enough detail so that a client-programmer can use it in an animation — a thorough description of its design and implementation is presented in Chapter IV of *Graphical Representation of Parallel Algorithmic Processes*[18].

The source for the parallel views library is located in the *PViews* subdirectory in the AAARF distribution and contains the following files:

- *PControl.c* Contains the functions that control the background process.
- *PViews.c* Contains the functions that process the incoming performance data and maintain the local data structures used by the performance views. Also contains the functions that control the appearance of the performance views. This file has an associated header file called *PViews.h*.
- *PVXXXX.c* There are ten of these files, each one containing the functions that draw and update a performance view.
- *Utils.c* This is a collection of miscellaneous functions that are needed to process data coming from the remote system.
- *PRASEBG.c* This is the background process. It serves as the link between the algorithm on the remote host and the views.

There is also a header file to be included in the program using the library in the *include* subdirectory:

- *PRASE.h* This header file contains the definitions needed by the client-programmer when using the library.

The externally useable functions are contained in the files *PControl.c*, *PViews.h*, and *Utils.c*; the rest of the files are described as well because they could serve as patterns or templates for custom views developed for an algorithm. Each of the files is discussed in the following sections.

6.1 PControl.c

The functions in this module are used mostly by the functions in the files *Control.c* and *View.c* in the *Skeleton* subdirectory. The prototypes for these functions are listed in Table 25. They are provided to centralize the interface with the background process in one place, and to put some of the performance items on the Master control panel.

termAlgProcessing() is a replacement for the function normally found in the *Control.c* class-specific file. It is provided here so that the implementation of the background process is insulated from the client-programmer.

resetPChild() is called at the beginning of the class-specific function *runAlgorithmInBackground()* in *Control.c*. It checks to see if the background process is active — if it is, the function sends the background process a reset command; otherwise, this function creates the background process. Initialization data is then sent to the background process.

getIE() is also a replacement for the class-specific function in *Control.c* of the same name. This function requests a trace block from the background process, and then waits until the trace block is received before returning.

getData() retrieves an algorithm data block for the specified node from the background process; a request is sent and the function returns when the data block is received.

setPViewItem() is called by the class-specific function *setViewItem()* in the file *Control.c*. It adds the performance views to the list of available views for the animation. This function should be called before any custom views are added because the constants defined in *PRASE.h* that are required to use the windows are very order-dependent.

setPBreakPointItem() is called by *setBreakPointItem()* in file *Control.c* in the *Skeleton* subdirectory. Its function is to add any performance related breakpoints to be added to the Master Control Panel; currently, there are no breakpoints available — the function is provided for completeness.

void	termAlgProcessing(void)
void	resetPChild(char *, char *, char *)
TRACE_DATA	*getIE(void)
char	*getData(int)
int	setPViewItem(Panel.item *)
void	setPBreakPointItem(Panel.item *)

Table 25: Function Prototypes for *PControl.c*

6.2 PViews.c

The functions in this file (listed in Table 26) have four tasks:

- Process the incoming data into data structures required by the displays
- Call the appropriate functions to draw and update the active displays
- Update the status panel
- Maintain the parameters for the performance views

`processPIE()` processes the trace data for the performance displays. It is called by `processIE()` in the file `View.c`. This function keeps track of the current simulated time and the times of the incoming records; if the time of the new record is in advance of the current simulated time, it returns the `ANIM_WAITING` value, which is passed back to `animateTheAlgorithm()` to prevent another trace block from being fetched until the current time has caught up with the time of the current record. This arrangement makes it necessary to do some processing in `processIE()` so that the `ANIM_WAITING` value is incorporated into its return value.

int	processPIE(int, TRACE_DATA *, int *)
void	resetPViews(void)
void	paintAllPViews(void)
void	updateAllPViews(int)
void	buildPStatusDisplay(Panel, int)
void	updatePStatus(void)
int	addPControlSection(Panel, int)
void	PPanelDispatch(Panel_item, Event *)
int	addPControlPanel(Panel, int)
void	PControlDispatcher(Panel_item, int, struct inputhead *)
Panel_setting	PTimeDispatcher(Panel_item, Event *)
void	PcontrolEventDispatch(Panel_item, Event *)
void	getPControlParameters(PARAMS)
void	setPControlParameters(PARAMS)
void	controlHelp(void)

Table 26: Function Prototypes for `PViews.c`

`resetPViews()` is called by `resetPChild()` to clear the local data structures before starting a new run of the animation.

`paintAllPViews()` and `updateAllPViews()` control the displays, calling the appropriate `paintXXXXview()` or `updateXXXXview()` function to perform the necessary operations.

`buildPStatusDisplay()` adds the performance trace information to the status panel. It is called by `buildStatusPanel()` in the file *View.c*, usually after the algorithm-specific information is added. `updatePStatus()` is called by `updateStatus()` to update the information in the status panel.

The performance views have a number of parameters that control what information is displayed and how it is displayed. To give the end-user access to these parameters, an item is added to the Master Control Panel (MCP) by `addPControlSection()`. This function is called by `addInputSection()` in the file *Input.c*. The parameters themselves are much too numerous to add to the MCP, so another panel is created to hold them — this panel is activated by a button added to the MCP. This control panel is shown in Figure 11. `PPanelDispatch()` is called by the Notifier when the button is pushed to activate the control panel.

`addPControlPanel()` is the function that creates the control panel. The functions `PControlDispatcher()`, `PTimeDispatcher`, and `PcontrolEventDispatch()` are called by the Notifier to handle user actions on the control panel. `controlHelp()` displays help for the control panel.

`getPControlParameters()` and `setPControlParameters()` are called by `getInputParameters()` and `setInputParameters` respectively to add the parallel view parameters to the collective state of the animation that is saved and restored by the common library.

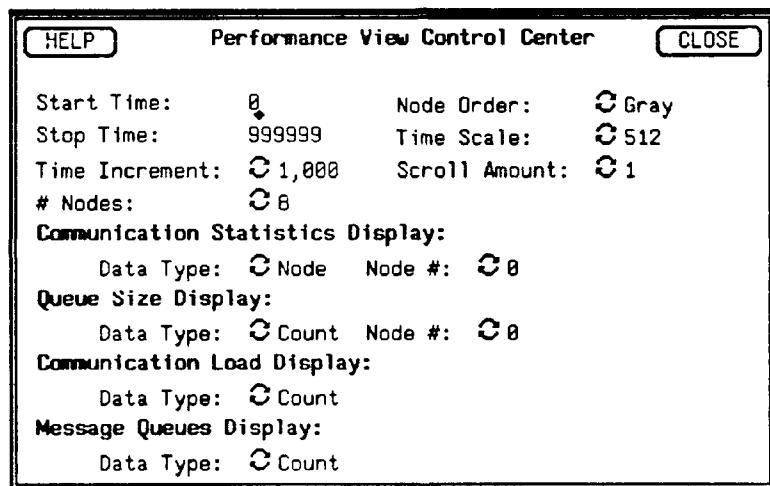


Figure 11: Performance Views Control Panel

6.3 PVXXXX.h (Draw and Update View)

There are ten of these files — one for each performance view:

- **ANIM** — Processor animation view
- **COMM** — System communications load view
- **CSTA** — Communications Statistics for a single processor
- **FEYN** — Feynman view of processor communications
- **GANT** — Traditional Gantt chart
- **KIVT** — Kiviat chart showing processor utilization
- **MLTH** — View of lengths of messages being passed
- **MSGQ** — Size of input queue for each processor
- **QSIZ** — Size of the input queue for one processor over time
- **UTIL** — Processor utilization view

The prototypes for the functions that may be found in these files are contained in Table 27.

Each view has a function to draw the display on the canvas (`paintXXXXview()`) and a function to update the display in response to new data or an advance in time (`updateXXXXview()`). If a view has local data structures that need to be updated, a `processXXXXdata()` function can be provided to accomodate the requirement. If the internal data structures of the view need to be cleared between animation runs, a `resetXXXXview()` can be included, and a call to this function is placed in the function `resetPViews()`.

```
void paintXXXXview(int)
void updateXXXXview(TRACE_DATA *, Pixwin *)
void processXXXXdata()
void resetXXXXview()
```

Table 27: Function Prototypes for a typical *PVXXXX.c*

6.4 Utils.c

This file contains miscellaneous functions to assist the client-programmer in dealing with data generated by the algorithm on the remote system. The prototypes for these functions are in Table 28.

`ginv()` is a function that accepts a node number and converts it to the node's position in a gray-ordered ring on a hypercube.

`Swap4()` performs the byte-order swapping necessary to convert integers from the Intel 'low byte first' arrangement to the Sun/SPARC 'high byte first' order.

int ginv(int)
int Swap4(short *, unsigned)

Table 28: Function Prototypes for *Utils.c*

6.5 PRASEBG.c

This is the background process that communicates with the remote system and the window-based process. A client-programmer doesn't explicitly call any function within this module, but if the rest of the library is used, this program is automatically invoked.

Since the client-programmer doesn't use this module directly, a full description of each function contained within it is not included here — a description of the design and implementation of this program is contained in Chapter IV of *Graphical Representation of Parallel Algorithmic Processes*[18].

The background process has these major tasks:

1. Receive and buffer event data coming from the remote system
2. Respond to data requests from the window-based process
3. Relay algorithm control requests to the remote system

To accomplish these tasks, the background process communicates with two programs that reside on the remote system: `aaarf_clct` and `server`. These programs are described in Section 7. The background process receives the event data from `aaarf_clct`, and sends the algorithm control commands to `server`.

6.6 PRASE.h

This file contains definitions that may be necessary when using the parallel views library. The most important item in the file is the structure definition of the trace data record that is received from the background process. This structure is shown in Table 29.

Except for the structure definition, this header file has the same format as the *Class.h* header file in the *Skeleton* subdirectory. Care must be taken, however, when assigning constants for views, breakpoints, and IE numbers — some of these values are already being used by the Parallel views. The *Class.h* file in the *Performance* subdirectory has comments inserted that indicate which values are being used by the library.

```
typedef struct
{
    int type;
    int IE_type;
    int node;
    int pid;
    unsigned long time;
    unsigned long busy;
    int dest;
    int dest_pid;
    int from;
    int from_pid;
    int msg_type;
    int msg_length;
    union
    {
        char mchar[15];
        int mint;
        long mlong;
        short mshort;
        double mdouble;
        float mfloat;
    } data;
}TRACE_DATA;
```

Table 29: Trace Data Structure

7 Parallel Program Instrumentation

The purpose of this section is to familiarize the client-programmer with the operation of the instrumentation that resides on the remote system. A detailed description of the design and implementation of these programs can be found in *Graphical Representation of Parallel Algorithmic Processes*[18].

The instrumentation consists of three components:

- Functions that are inserted into the user's program to gather data
- The program `aaarf_clct` that collects the data and passes it to the background process on the workstation
- The program `server` that can start and stop the user's program

7.1 Instrumentation Functions

These functions are contained in the library `aaarf_inst.a`. They, with the help of the PRASE preprocessor[7], intercept calls to system routines so that data can be collected about them. The performance data that is collected is sent to the `aaarf_clct` program executing on the host processor of the hypercube. The instrumentation routines are slightly modified versions of the PRASE instrumentation functions[6]. These are the functions contained in the library:

- System call intercept routines:

- `praseexit()`
- `prase_exit()`
- `praseflick()`
- `praseled()`
- `praseiprobe()`
- `praseirecv()`
- `prasecrecv()`
- `praseisend()`
- `prasecsend()`

- User data routines:

- `prasemarkchar()` Inserts a short string into the trace data stream.

- **prasemarkdouble()** Inserts a double precision number into the trace data stream.
- **prasemarkfloat()** Inserts a single precision number into the trace data stream.
- **prasemarkint()** Inserts an integer number into the trace data stream.
- **prasemarklong()** Inserts a long integer number into the trace data stream.
- **prasemarkshort()** Inserts a short integer number into the trace data stream.

- Instrumentation support routines:
 - **praseinit()** This function initializes tracing and synchronizes the nodes so that all times are based on a common reference point.
 - **prase_dump()** This function is called by other instrumentation routines to send a record.
 - **praseend()** Called by the main function just before termination, this function inserts an *end* record into the trace data.
 - **prase_sync()** This function is called by **praseinit()** to synchronize the nodes.

The only functions that the client-programmer would use (if any of these are used) are the user data routines — the rest are automatically invoked at the appropriate time.

The user data routines are used to pass algorithm data through the instrumentation to the workstation for use in creating and updating displays. There are two functions that the client-programmer calls to pass algorithm data:

IE(type, data1, data2) This function is used to pass small amounts of data — specifically, two integers and the event type. This event is passed to the display system using the **prasemarkchar()** instrumentation routine; the integers are formatted into a string and put into the trace record.

IE_data(type, pointer, length) This function is used when there is more data to be passed than **IE()** can hold. The pointer can point to any data block of any length (up to 128k bytes). The event type is sent using the **prasemarkint()** instrumentation routine, and the data block is sent as it is to **aaarf_clct**.

7.2 aaarf_clct

This program is a separate process that executes on the host processor of the hypercube while the user's program is running on the node processors. It listens for data messages being sent from the nodes, arranges them in time order, and relays them to the background process on the workstation.

7.3 server

This is the program that controls the user's program in response to commands sent from the background process on the workstation. It receives two parameters from the background process when the user's program is to be started: the directory to make the default for running the user's program, and the complete command line necessary to invoke the user's program. When the command is received to terminate the user's program, it issues the appropriate commands to carry out the request.

8 Animating an Algorithm

This section suggests a procedure that can be followed when animating an algorithm. This procedure can be applied equally to sequential and parallel algorithms — the main difference is the actual methods used to extract the information from the program.

8.1 Animation Process

The process of animating an algorithm can be decomposed into three tasks:

1. Analyze the algorithm to determine the basic operations
2. Develop ways of displaying the algorithm
3. Instrument the program

These steps assume that the algorithm has already been implemented to the point that the program can be executed on the target system.

Even though the process has been divided into three parts, there isn't a sharp distinction between them. The analysis of the algorithm needs to be done with the displays in mind. When developing displays, the side effects of instrumentation need to be considered. Instrumentation must be done so that the displays have all the data needed so they can function. The interrelationship is so tight that all three need to be worked concurrently. There is no specific order in which things must be done. It is an iterative (and concurrent) design process that only stops when the programmer is satisfied with the visualization (animation) results.

8.1.1 Analyzing the Algorithm

In order to animate an algorithm, it needs to be analyzed to distill its basic components. These basic components fall into two categories: operations and data structures. The operations are also referred to as *events*. “Animating the algorithm consists of developing meaningful graphical representations of the algorithm events and intermediate states[4:12]”. Simply displaying the data structures and their contents is not enough since frequently algorithms don't modify the contents of their data structures at all! The actions taken by the algorithm and the decisions made by the algorithm can present considerably more about an algorithm than just the data structures.

The analysis of the data structures should be focused on what data is necessary to display not only the external results of the algorithm (an “external” view), but

also some of the internal operations ("internal" views). In some cases, the primary data structure used for display is not even used in the algorithm!

In the context of algorithm animation, there are two types of algorithm events: events that correspond to actual algorithm operations and events that provide information to the user or the display program. The latter type of events may include notification that a certain phase of operation is complete, or simply passing data structures to the display program to use for the displays.

The events must be chosen such that the display can be updated or modified when something "noteworthy" occurs in the algorithm. Sometimes this means choosing an event that signals the start of a process, and one that signals the end. The purpose of the event at the end may be simply to pass an updated data structure to the display, or to reset the display to keep it from getting cluttered. Events should also be identified whenever one of the significant data structures changes, which allows the displays to keep track of the current state of these structures.

The choice of events is the very core of the animation design — the rest of the animation is driven by the events produced during a run of the algorithm. These guidelines can be used to determine what events to use:

Using the program source code as implemented on the target system (or even a higher level design language), one should identify the major operations performed by the algorithm:

- Start at a high level to get the big picture, and then go to lower levels to get more detail if needed.
- Function calls in the top-level routine are prime candidates for events.
- If the program has several phases of operation, consider events that mark the transition between phases.

Remember that the level of detail needed is driven by the complexity of the displays — some displays are very effective with mainly high-level events, while others need a detailed account of the algorithm's progress.[18:5-6]

8.1.2 Displaying the Algorithm

This activity is in some ways the whole purpose of the animation effort. It is definitely the end result of the animation design process. This is also the most difficult step — there are no specific methods to generate displays for the algorithm data. There may also be several different ways that the same data can be displayed:

In an ideal program analysis environment, each aspect of ... program behavior would be revealed by some view wherein the cause is manifest. No

one view is sufficient; any particular view contains either too much or too little information to understand some aspects of a program's behavior. Unless the programmer knows exactly what to look for, interesting phenomena can be lost in the sheer volume of information.[9]

The developer of the displays relies upon creativity and experience to determine appropriate ways of displaying the data for the given application.

8.1.3 Instrumenting the Program

This is where the animation process differs for sequential and parallel algorithms; the method used to transfer the data from the algorithm to the display is highly dependent on the environment in which each part is executing. The function calls may look similar, but there is much more that needs to be done for parallel algorithms.

For each of the events chosen, insert a call to the data collection function. There are three general situations that occur when inserting the instrumentation calls:

- The event is simply a marker, with no data
- The event requires data, and the data is already available
- The event requires data, and the data does not exist

The first case is straightforward: insert a call containing the event number. The next situation is almost as simple — insert a call containing the event number and the associated data. The last situation occurs when the event number or event data is not explicitly available from variables or location within the program. In this case, statements must be inserted into the program to determine the data or event number to send. For performance and efficiency reasons, this should be avoided, since the calculations needed to generate the event data can affect the operation of the algorithm; In some cases, though, it is the only way to get certain types of data.

Sequential Algorithms For sequential algorithms, the only data that needs to be collected is that which is required for the displays. This is accomplished by adding a function to the program similar to the `IE()` function in `BG.c` in the *Skeleton* subdirectory. Modify the function so that it collects and sends the data provided by the algorithm. The data sent is usually defined by a structure definition in `Class.h` so that all the class specific modules have access to it.

Parallel algorithms With parallel programs, instrumenting the target program occurs in two stages — first, the performance monitoring routines are inserted, then the function calls to extract the algorithm data for the events. The performance

```

#define PRASE

#include "aaarf_incl.h"

extern PROC prase_procs[MAX_NODES];

extern unsigned long prase_start_time;
extern unsigned long prase_lowest_node;

#define exit      praseexit
#define _exit     prase_exit

#define flick    praseflick
#define irecv    praseirecv
#define crecv    prasecrecv
#define isend    praseisend
#define csend    prasecsend

#endif

```

Figure 12: Typical instrumentation header file

instrumentation is inserted first for two reasons: performance data is normally needed for “complete” analysis of the algorithm, and the event data extraction functions use the performance instrumentation to carry the events to the *display system*.

Because of the existence of the PRASE preprocessor[7:3-1], inserting the performance instrumentation is simple; the programmer creates a configuration file that describes the program[7:2-3], and the preprocessor does the rest. If for some reason the instrumentation needs to be inserted manually, it is a simple process: a header file is included at the top of every source file, and some code is inserted at the very beginning and end of the main process on the node. The header file is shown in Figure 12 and the additions to the main function are shown in Figure 13.

There are two pieces of code that need to be inserted into the host process — the aaarf.clct process must be started at the beginning of the run before the node processors are loaded, and the host process must wait for all the trace data to be collected before it terminates the node processes. These code fragments are shown in Figure 14.

Inserting the instrumentation for all the algorithm data must be done manually. There are two methods for marking an event, depending on how much data is required to accompany the event. If there is little or no data, the IE() function is used; it accepts three parameters — the integer event number and two integer data values. If there is more data associated with the event than will fit in the IE() function call, there is another function called IE_data() that accepts the integer event number, a pointer to a data block, and the length of the block. The IE_data() function is

```

main()
{
    /* local variables */

#ifndef PRASE
    prase_procs[0].num_pids = 1;
    prase_procs[0].pids[0] = 0;
    prase_procs[1].num_pids = 1;
    prase_procs[1].pids[0] = 0;
    prase_procs[2].num_pids = 1;
    prase_procs[2].pids[0] = 0;
    prase_procs[3].num_pids = 1;
    prase_procs[3].pids[0] = 0;
    prase_procs[4].num_pids = 1;
    prase_procs[4].pids[0] = 0;
    prase_procs[5].num_pids = 1;
    prase_procs[5].pids[0] = 0;
    prase_procs[6].num_pids = 1;
    prase_procs[6].pids[0] = 0;
    prase_procs[7].num_pids = 1;
    prase_procs[7].pids[0] = 0;

    prase_lowest_node = 0;
    prase_start_time = 0;

    praseinit();
#endif

    /* normal processing */

#ifndef PRASE
    praseend();
#endif

    /* termination processing */
}

```

Figure 13: Instrumenting the main node function

```
main()
{
    /* local variables */

#ifndef PRASE
    FILE *prase_ptr;
#endif

    /* start of main program */
    /* initialization */
    getcube(...);

#ifndef PRASE
    system("aaarf_clct 1000000 &");
#endif

    /* load node programs */
    /* process data */

#ifndef PRASE
    praseend();
#endif

    /* terminate node programs */
}
```

Figure 14: Instrumenting the host process

slower, and should be used only when necessary.

8.1.4 Iteration

As was stated previously, the process of animating an algorithm is an iterative activity, somewhat similar to the spiral lifecycle model for software development[1] — start out completing a small-scale implementation or prototype, then add to it in stages to develop the final implementation.

The events that are chosen depend to some extent on what is needed to create the displays; the development of the displays depends on the types of data that can be extracted without imposing an unacceptable performance penalty. The instrumentation task is the only one that truly must be done as the last step in an animation development cycle.

9 AAARF Projects

This section presents suggestions for AAARF extensions and animation projects.

1. Extend the AAARF class file to include other configuration variables that are currently either compile time defines, hard-coded defines, or environment variables. For example, maximum number of windows; maximum number of views; maximum and minimum window size; default recording and environment paths; default algorithm window color codes; etc. Most likely, two configuration files are required: one for the AAARF administrator and one for the user.
2. Implement a "snap" option which forces the width, height, and position of windows to be a multiple of some number of pixels. This feature makes it easier to create windows of the same size and to align them neatly for comparisons.
3. Modify the animation recorder such that reverse playback is supported. This allows the end-user to immediately review an interesting part of a (recorded) animation, without restarting the recording.
4. Extend the animation recorder's capabilities such that every animation is always recorded. This allows the user to use the reverse button at any time during the animation. An interesting part of an animation could be reviewed immediately (currently, the user must wait for the algorithm to finish before playing the recording). When the user hits the reverse button, the recorder provides IEs to the view functions; the background procedure continues to wait with its next IE. If the animation returns to the point at which the reverse button was hit, IEs are again retrieved from the background procedure.
5. Extend the capability of the recorder such that it records control events. For example, detect and record speed changes, changes in the break point setting, user requested stops, etc. This permits end-users to create recordings that emphasize a particular event or set of events within an animation.
6. Develop a linkable library of animation objects similar to that developed by John Stasko in TANGO [10]. The library should provide a set of primitive graphic objects and functions to support smooth animation of the objects between two points or along a predetermined path. This could greatly simplify the development of the view functions, arguably the most grueling programming task for the client-programmer.
7. Modify the AAARF class-common library such that it is a linkable library.

8. Add a *current state* indicator to the algorithm window structure. The indicator reflects the current state of the algorithm (running, stopped, finished, etc) and the current state of the recorder (recording, reverse, forward, paused, etc). Unlike the status display panel which may be opened or closed, this indicator is always displayed. The information can appear on the title bar or within some reserved area in the algorithm window.
9. Modify AAARF's color table usage so that pixrect patterns are used in addition to color. This should provide more interesting displays on monochrome monitors.
10. Develop a bin sorting algorithm class. Algorithms should include first-fit, best-fit, worst-fit.
11. Develop a tree searching program that animates various type of search algorithms: depth-first, breadth-first, etc.
12. Animate several numerical approximation methods such as finding roots, integration, least common denominator.
13. Animate the "Drinking Philosophers" problem using a petri net representation.

References

- [1] Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes*, 11(4):11-14 (1986).
- [2] Brown, Marc H. *Algorithm Animation*. Cambridge, Massachusetts: The MIT Press, 1987.
- [3] Fife, Keith C. "The AAARF User's Manual." Air Force Institute of Technology, November 1989.
- [4] Fife, Keith C. *Graphical Representation of Algorithmic Processes*. MS thesis, Air Force Institute of Technology, 1989.
- [5] Hartrum, Thomas C. *System Development Documentation Guidelines and Standards*. Draft #4, Air Force Institute of Technology, January 1989.
- [6] Kahl, Mark Albert. *PRASE: Instrumentation Software for the intel iPSC Hypercube*. MS thesis, Air Force Institute of Technology, 1988.
- [7] Kahl, Mark Albert. "The PRASE User's Manual." Air Force Institute of Technology, December 1988.
- [8] Kernighan, Brian and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Inc, 1978.
- [9] LeBlanc, Thomas J., et al. "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, pages 203-217 (September 1990).
- [10] Stasko, John T. "Simplifying Algorithm Animation Design with TANGO." Georgia Institute of Technology, June 1989.
- [11] Sun Microsystems. *C Programmer's Guide*, 1988. SunOS Technical Documentation.
- [12] Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.
- [13] Sun Microsystems. *Network Programming*, 1988. SunOS Technical Documentation.
- [14] Sun Microsystems. *Pixrect Reference Guide*, 1988. SunOS Technical Documentation.

- [15] Sun Microsystems. *Setting Up Your SunOS Environment: Beginner's Guide*, 1988. SunOS Technical Documentation.
- [16] Sun Microsystems. *SunView1 Beginner's Guide*, 1988. SunOS Technical Documentation.
- [17] Sun Microsystems. *SunView1 Programmer's Guide*, 1988. SunOS Technical Documentation.
- [18] Williams, Edward M. *Graphical Representation of Parallel Algorithmic Processes*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1990.

Appendix D. *AAARF User's Manual*

This appendix contains the AAARF User's Manual. The changes made to AAARF did not affect the existing user interface, but the new capabilities needed to be included. The entire manual is contained in this appendix, even though only portions were modified. The original manual was written by Keith Fife [11].

Since the manual is intended to be a separate document, its page numbers do not follow the numbering system for this thesis.

AAARF User's Manual

Keith Carson Fife
Captain, USAF

Edward Michael Williams
Captain, USAF

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45432

December, 1990

Contents

1	Introduction	4
2	Overview	6
2.1	AAARF Architecture	7
2.2	Windows	9
2.3	The Mouse	12
2.4	Menus	13
2.5	Panels	13
2.6	Alerts	16
3	Getting started	17
3.1	Welcome Screen	17
3.2	The Main Menu	17
3.3	Algorithm Class File	19
3.4	Central Control Panel	19
3.5	Environment Control Panel	20
4	Algorithm Windows	22
4.1	Algorithm Window Menu	22
4.2	Master Control Panel	24
4.3	Animation Recorder	25
4.4	Status Display	28
5	Parallel Algorithm	29
5.1	Remote System Set-up	29
5.2	Running the Animation	29
5.3	Parallel Performance Views	30
6	Exercises	39

List of Figures

1	Algorithm Animation Components	6
2	AAARF Levels of Execution	8
3	Types of Windows used by AAARF	9
4	Multiple Algorithm Windows	10
5	Multiple Views of a Single Algorithm Animation	11
6	Mouse Button Usage	12
7	AAARF Menus	13
8	Panel Item Examples	14
9	Manipulating Windows and Panels	15
10	AAARF Alert Example	16
11	AAARF Main Menu	18
12	AAARF Environment Control Panel	20
13	Algorithm Window Menu	23
14	Master Control Panel for Traversal Algorithm Class	24
15	AAARF Animation Recorder	26
16	Status Display for ArraySort Algorithm Class	28
17	Sample Utilization View	31
18	Sample Gantt View	32
19	Sample Feynman View	32
20	Sample Communications Statistics View	33
21	Sample Communications Load View	34
22	Sample Queue Size View	34
23	Sample Message Lengths View	35
24	Sample Message Queues View	36
25	Sample Kiviat View	37
26	Sample Animation View	38
27	Possible Window Arrangement for Exercise 1	40

Revisions

December 1989 This document was first released by Capt Keith Fife as a part of his Master's Thesis[3].

December 1990 Section 5 was added to document the enhancements made by Capt Ed Williams to support research for his Master's Thesis[8]. The major changes are:

- Section 1 is updated to describe the new material
- Section 5 is added to describe the use of AAARF to animate parallel processes

AAARF User's Manual

Captain Keith C. Fife

Captain Edward M. Williams

1 Introduction

The AFIT Algorithm Animation Research Facility (AAARF) is an interactive algorithm animation system. It provides a means for visualizing the execution of algorithms and their associated data structures. AAARF allows the user to select the type of algorithm, the input to the algorithm, and the views of the algorithm. Several control mechanisms are provided, including stop, go, reset, variable speed, single-step, and break-points. Other features of AAARF include:

- Multiple Algorithm Windows
- Simultaneous Control of Multiple Animations
- Animation Environment Save and Restore Capability
- Multiple View Windows within each Algorithm Window
- Animation Record and Playback
- Algorithm State Display and Interrogation Capability
- Master Control Panel for monitoring and modifying the input, algorithm, view, and control parameters to an algorithm animation.

AAARF runs on Sun3TM and Sun4TM workstations using the SunOSTM (Sun Microsystem's version of the AT&T UNIXTM operating system) and the SunViewTM window-based environment. AAARF is designed for use with color monitors. Monochrome monitors can be used, but the displays are not as informative.

AAARF has two types of users: *end-users* who view and interact with the algorithm animations, and *client-programmers* who develop and maintain the algorithms and animations. This manual is primarily for end-users; it describes how to use AAARF. Client-programmers should refer to the *AAARF Programmer's Manual* [2].

This manual introduces users to algorithm animation and AAARF. It explains how AAARF can be used to explore algorithms in ways not previously possible. No previous experience with algorithm animation is required; interested readers may refer to *The Graphical Representation of Algorithmic Processes* [3] for more detailed information. Though not necessary, some familiarity with SunOS and SunView is helpful. The following references are recommended:

- *Getting Started with SunOS: Beginner's Guide* [5]
- *Setting Up Your SunOS Environment: Beginner's Guide* [6]
- *The SunView1 Beginner's Guide* [7]

The next section introduces algorithm animation, describes the AAARF system architecture, and presents an overview of AAARF's use of SunView. Users should be thoroughly familiar with the concepts presented in this section before using AAARF. Section 3 explains how to start the AAARF program and describes the major components of the AAARF main screen. Section 4 discusses algorithm windows and explains how to start and control animations. Section 5 describes the steps that need to be taken to run animations of parallel programs. Section 6 presents some exercises to test and develop an understanding of AAARF.

2 Overview

This section examines the general architecture of AAARF and introduces some SunView concepts necessary for understanding and effectively using AAARF. The definitions that follow are extracted from [3] and [1]; they are critical to an exact understanding of the concepts of this section:

Animation Components An algorithm animation consists of three components: an input generator, an algorithm, and one or more animation views (see Figure 1).

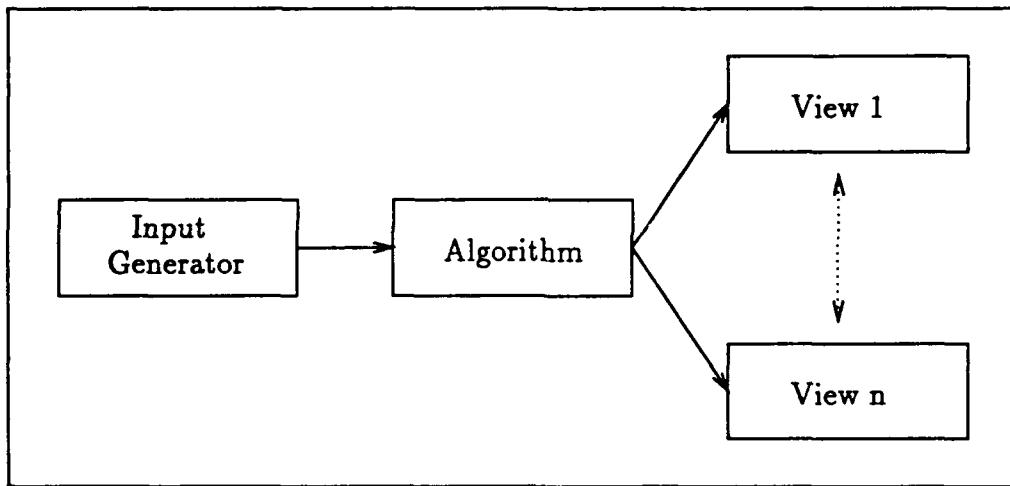


Figure 1: Algorithm Animation Components

Input Generator An input generator is a procedure which provides input to an algorithm; the input may be generated randomly, read from a file, or entered by the user.

Animation View Animation views are graphical representations of an algorithm's execution.

Interesting Event (IE) Animation views are driven by interesting events that occur during the execution of an algorithm. Interesting events are input events, output events, and state changes that an algorithm undergoes during its execution. The type, quantity, and sequence of IEs for a particular algorithm distinguish it from other algorithms.

Algorithm Classes Algorithms which operate on identical data structures and perform identical functions are from the same algorithm class. The *Array Sort* class includes *quick sort*, *heap sort*, *bubble sort*, and other in-place sort algorithms. Algorithms from the same class generally share input generators and views, although certain views and input generators may be ineffective with particular algorithms. For instance, the *tree* view is very meaningful for heap sort, but nearly useless for any other sort.

Parameterized Control User-selectable parameters are associated with each component. *Algorithm parameters* affect some aspect of how the algorithm executes. For example, with a quick sort algorithm, what partitioning strategy should be used; as the partitions get smaller, at what point should another type of sort be used; what other type of sort should be used. *Input Parameters* affect the input generator — what seed is used to generate a set of random numbers; how “sorted” is a set of unsorted numbers; what is the general form of a series of numbers. *View parameters* affect how the animation is displayed in the view window. For example, what shape is associated with the nodes in a graph; how should an arbitrary graph be positioned.

2.1 AAARF Architecture

AAARF uses three levels of execution linked via UNIX sockets to animate algorithms. Figure 2 shows the levels of execution.

AAARF Main Process.

AAARF’s top-level process provides the main screen, a mechanism for saving and restoring animation environments, a mechanism for controlling multiple algorithm animations simultaneously, and a means for starting new algorithm animations. This level serves as a high-level manager of the animation environment.

- *In general, The AAARF main process is the only level of execution with which end-users need be concerned.*

Class-Specific Window-Based Process.

The second level of execution is the algorithm class-specific window-based process. The class-specific level of execution provides the animation views, an animation recorder, a status display for interrogating the state of the algorithm, and a master control panel for monitoring and modifying the parameters that control the animation.

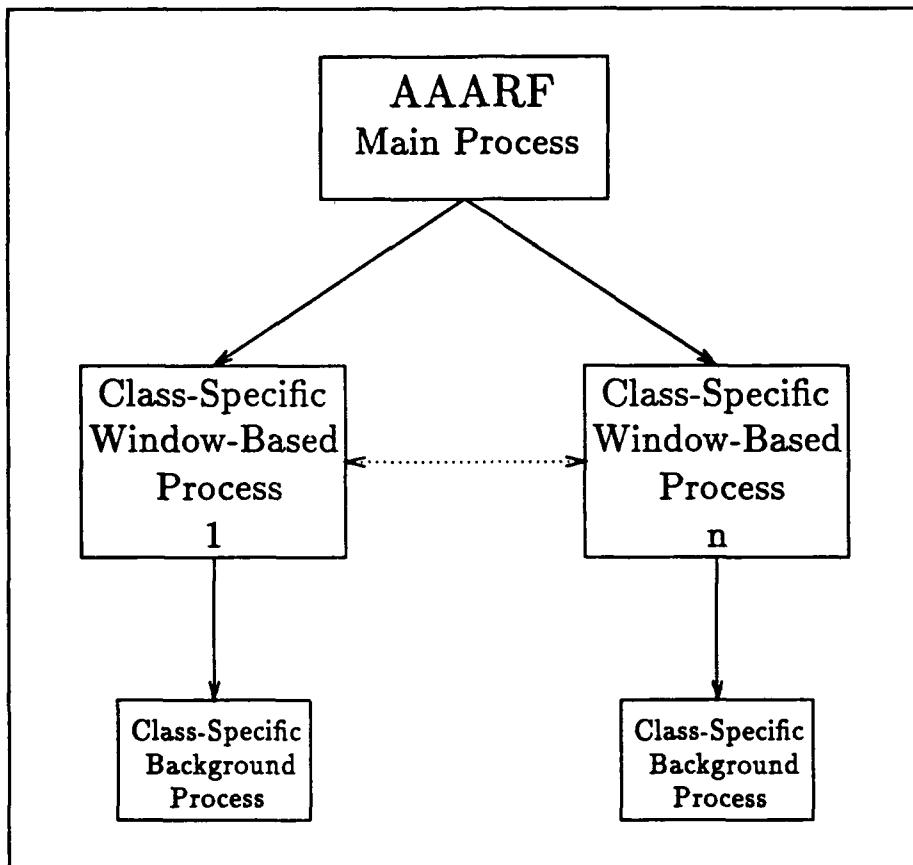


Figure 2: AAARF Levels of Execution

Class-Specific Background Process.

The third level of execution is completely transparent to the user; this level implements the input generator and algorithm. It provides the window-based level with the IEs that drive the animation. The actual algorithms being animated run at this level. The background process waits with the next IE until the window-based level sends an IE request. The background process sends the IE and executes the algorithm until the next IE occurs. It stops again and waits for an IE request from the window-based level.

2.2 Windows

AAARF uses three types of windows to provide the user with multiple simultaneous algorithm animations and multiple views of each animation. Figure 3 shows the types of windows used by AAARF and their relationship to one another.

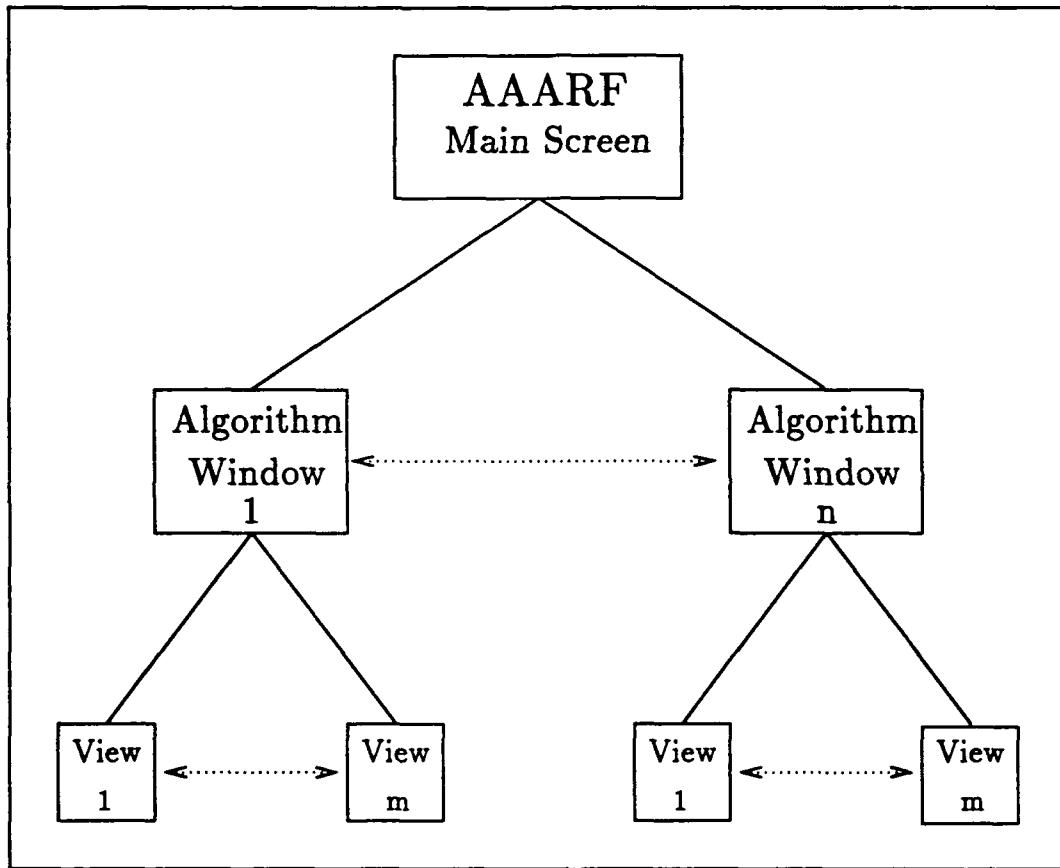


Figure 3: Types of Windows used by AAARF

AAARF Main Screen.

This is a full-screen window within which all interaction with AAARF is contained. The main menu can be *popped up* anywhere on the AAARF main screen. The main screen supports up to four algorithm windows.

- *The actual number of algorithm windows possible is dependent on the system configuration, cpu load, and memory availability.*

Algorithm Window.

All animations take place within algorithm windows. Every algorithm window is associated with a particular algorithm class. Algorithm windows may be created and destroyed freely, but no more than four can be active at any time. Each algorithm window supports an animation recorder, a master control panel, a status display, and up to four view windows. Algorithm windows can be resized and moved anywhere on the AAARF screen; they may overlap one another. The algorithm menu can be popped up anywhere along the algorithm window title bar.

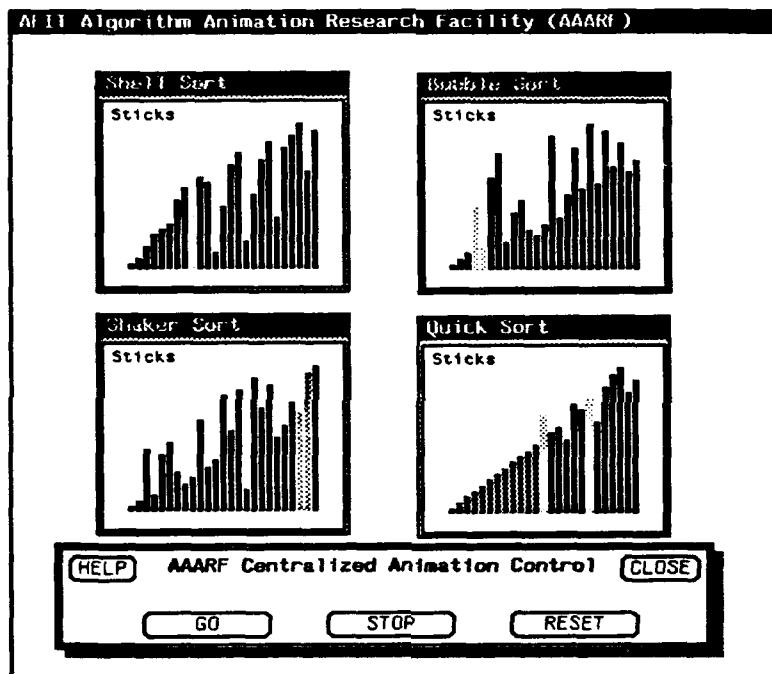


Figure 4: Multiple Algorithm Windows

View Windows.

View windows, or views, are windows associated with a particular view of an algorithm. Every algorithm window has at least one and as many as four active view windows. View windows can be resized and moved, but they cannot extend beyond the algorithm window and they may not overlap.

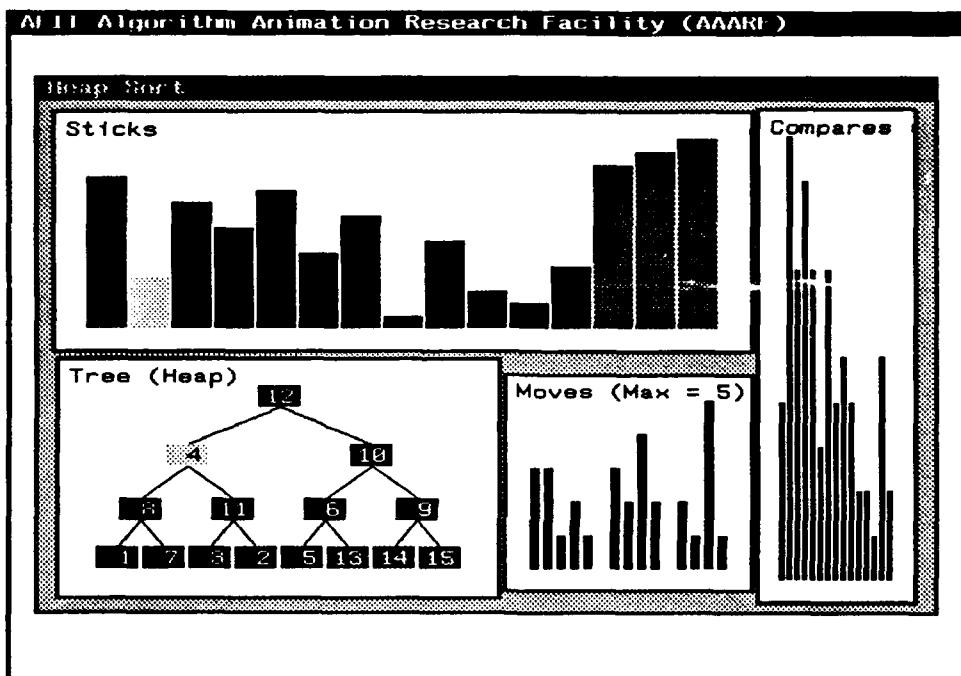


Figure 5: Multiple Views of a Single Algorithm Animation

- Right Button. The right button is used almost exclusively to pop up menus (see Section 2.4) from which selections can be made.
- Middle Button. The middle button is used primarily to position and resize windows. Figure 9 describes the procedure for moving and resizing windows. The middle button is also used to select an element of interest within a view window by clicking on the desired element.
- Left Button. The left mouse button is used to activate panel items (see Section 2.5). It is also used to control algorithm animations; clicking in a view window starts and stops the animation.

Figure 6: Mouse Button Usage

2.3 The Mouse

Almost all interaction with AAARF is through the mouse. The mouse is used to move the pointer, or cursor, around the screen. The mouse buttons may be either *clicked* (pushed and immediately released) or *depressed* (pushed and held until some action is complete). The function of the mouse buttons depends on the application; Figure 6 describes the mouse button functions. Figure 9 explains how to use the mouse to manipulate windows and panels.

- *Just as you can type ahead of the display with the keyboard, you can “mouse ahead” of the display with the mouse. Depending on the CPU load and the number of active processes, display update can be slow. Rest assured the mouse input will, eventually, be acknowledged. Be careful when mousing ahead – you may be clicking on something that won’t be there when the click is serviced.*

- Main AAARF Menu. This menu is accessed by pressing the right mouse button anywhere on the AAARF background screen.
- Algorithm Window Menu. This menu is accessed by pressing the right mouse button anywhere along the title bar of an algorithm window.
- Panel Selectors. These menus are accessed by depressing the right mouse button on panel selectors (Section 2.5) or panel choice items (Section 2.5).

Figure 7: AAARF Menus

2.4 Menus

AAARF uses menus to allow users to make a selection from among several choices. Users *pop up* menus by depressing the right mouse button. Generally, menus remain visible only as long as the right button remains depressed. While a menu is visible and the right mouse button is depressed, moving the cursor over a particular menu entry causes that entry to be highlighted. Releasing the right mouse button with a menu item highlighted *selects* that menu item. Figure 7 describes the three types of menus used in AAARF.

A menu entry with a right-arrow indicates that a *pull-right* menu is associated with that menu item. Moving the cursor over the right-arrow exposes the pull-right menu from which a selection can be made. Usually, the first item in a pull-right menu is the default selection for an item with a pull-right menu.

2.5 Panels

Panels allow users to interact with AAARF through a variety of *panel items* such as, *panel buttons*, *panel selectors*, *choice items*, and *panel text items* (see Figure 8).

Panel Buttons

Panel buttons are used to specify an action. Panel buttons are activated by clicking the left mouse button with the mouse pointer positioned over the panel button.

Panel Selectors

Panel selectors are used to pop up a menu from which a particular menu item can be selected. Panel selectors are activated by depressing the right mouse button with the mouse pointer positioned over the panel selector. The mouse button is released after the desired menu selection is highlighted.

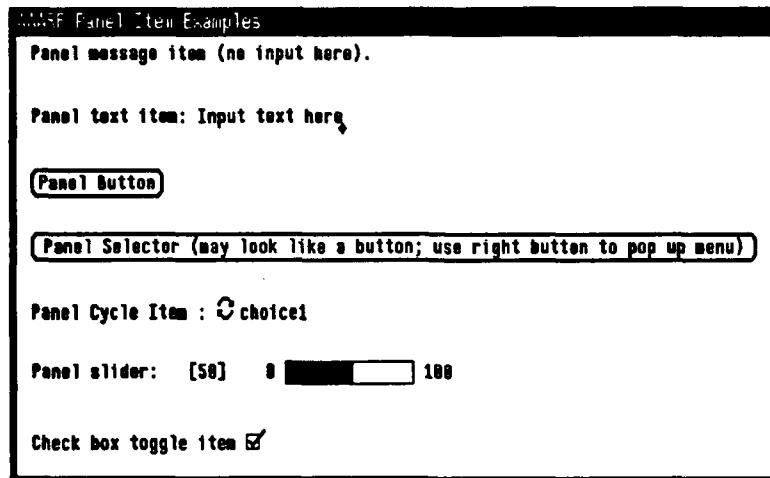


Figure 8: Panel Item Examples

Choice Items

Choice items are used to set options. Choice items appear as *cycle items*, *choice buttons*, *check boxes*, and *panel sliders*. All choice items except the panel slider can be used as either a panel button or a panel selector. Panel sliders are activated by clicking the left mouse button at the desired position on the slider bar.

Text Items

Text items allow the user to enter data from the keyboard. AAARF requires very little keyboard interaction.

- **Moving Windows and Panels.** To move a window, position the cursor over the window's border. Press and hold the middle button while repositioning the cursor to the window's new position. Release the middle mouse button and the window moves to the new location. Panels are moved with the same procedure.
- **Resizing Windows.** To resize a window, position the cursor over the window's border, press and hold the control key and the middle mouse button while repositioning the cursor to the window's new border position. Release the mouse button and the window resizes. Panels cannot be resized.
- **Bringing Windows and Panels to the Front.** To completely expose a partially hidden window, click the left mouse button on the hidden window's border. The procedure for panels is identical.

Figure 9: Manipulating Windows and Panels

2.6 Alerts

AAARF uses alerts to display help screens, to warn users of internal errors, and to get confirmation for dangerous commands. The user must click on one of the alert buttons before input is accepted in any other window or panel. Figure 10 shows an alert asking the user to confirm deletion of an environment file. Note that the YES button has a darker outline than the NO button. A button with a dark outline indicates that it is the default choice; it may be selected with the mouse or, alternatively, by hitting the return key.

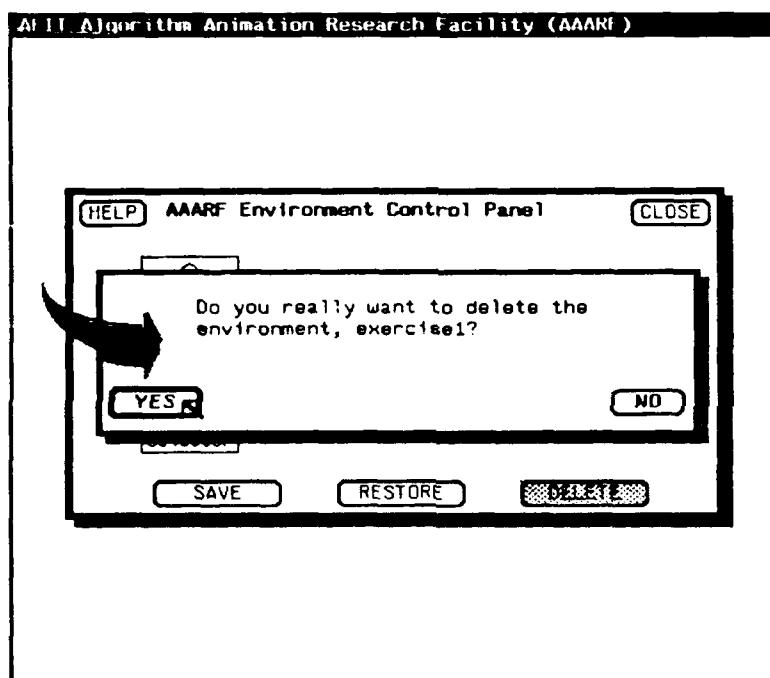


Figure 10: AAARF Alert Example

3 Getting started

Set your UNIX *path* variable to include the AAARF executable directory. Your instructor or system manager knows the AAARF directory path. See [6] for more information on setting UNIX paths. Run AAARF by entering *aaarf* at the UNIX command line prompt.

3.1 Welcome Screen

The first thing you *see* is the AAARF welcome screen. Click the left mouse button on the **CONTINUE** button to begin the session.

3.2 The Main Menu

Press and hold the right mouse button with the mouse pointer anywhere on the AAARF screen to pop up the main menu. Move the cursor to select from the following choices:

- Iconify AAARF
- New Algorithm Window
- Central Control
- Environment Control
- Help
- Kill AAARF

Iconify AAARF

This selection causes AAARF and all its animations to become an icon. The icon may be moved anywhere on the screen. While in the icon state, all animations are stopped and most menu items are deactivated. AAARF is deiconified by either clicking the right mouse button on the icon or selecting Deiconify AAARF from the main menu.

New Algorithm Window

To open a new algorithm window, highlight this selection. Move the cursor over the right arrow to reveal the algorithm class menu (see Section 3.3). Highlight one of the available classes and release the mouse button. Figure 11 shows the AAARF main menu with the algorithm menu exposed. An algorithm window with a default data

set and algorithm appears. Up to four windows may be active simultaneously. See Section 4 for controlling the algorithm window.

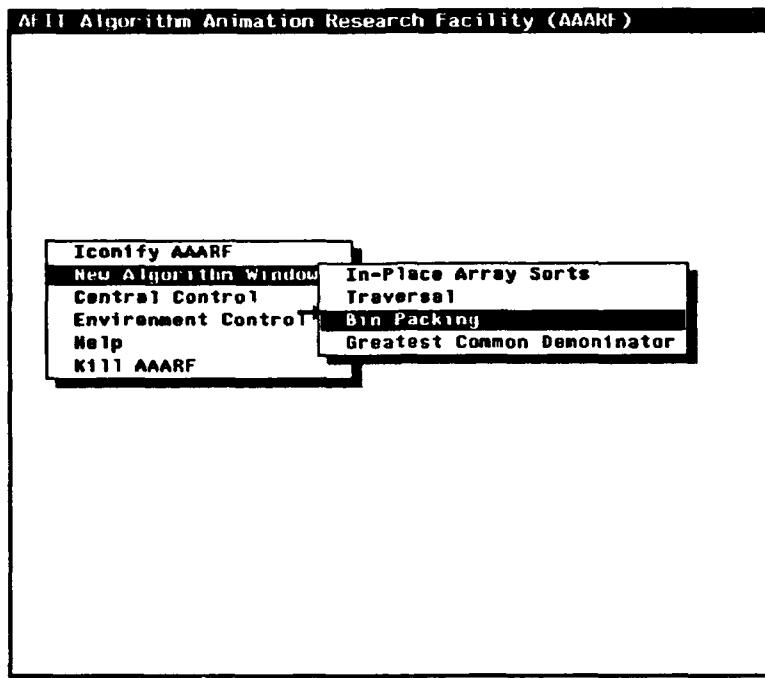


Figure 11: AAARF Main Menu

Central Control Panel

This selection causes the Central Control Panel to appear. The Central Control Panel provides a means to simultaneously control the animations in all open algorithm windows. See Section 3.4 for more information about the Central Control Panel.

Environment Control Panel

This selection causes the Environment Control Panel to appear. The Environment Control Panel provides a means to save and restore AAARF environments. See Section 3.5 for more information about the Environment Control Panel.

Help

This selection displays a help screen which briefly explains the function of each main menu item, how to use the mouse, and how to open a new algorithm window. This help screen is the same as the welcome screen that appears immediately after the program is evoked.

Kill AAARF

This selection ends the AAARF program. The user is first asked to confirm the kill command.

3.3 Algorithm Class File

AAARF has a default set of algorithm classes from which the user can select when opening new algorithm windows. The default classes are named in the *.aaarfClasses* file located with the AAARF executable image. The default algorithm class file can be overridden by setting the **AAARFCLASSES** environment variable to the filename of some other algorithm class file. See [5] for more information regarding UNIX environment variables.

- *The default algorithm class file suffices for most end-users.*

3.4 Central Control Panel

The AAARF Central Control Panel provides a means for simultaneously controlling multiple animations. The usual animation controls provided by the algorithm window are still enabled when using the Central Control Panel. Figure 4 shows the Central Control Panel. Use the right mouse button to initiate the following actions:

- **Help** Displays a help screen explaining the Central Control Panel function and controls.
- **Close** Closes the Central Control Panel. The panel can be reopened by selecting it again from the main AAARF menu.
- **Go** Simultaneously starts animations in all open algorithm windows. Animations which have already run to completion are not affected. Use **Reset** followed by **Go** to restart a completed animation
- **Stop** Simultaneously stops animations in all open algorithm windows. Has no affect on animations that have already run to completion
- **Reset** Simultaneously stops and resets all animations to their respective initial conditions. Animations may be running when the **Reset** button is pushed. There is no **UNDO** for a **Reset**.

3.5 Environment Control Panel

The AAARF Environment Control Panel provides a means for saving the current animation environment or restoring a saved environment. The environment includes *all* user options currently selected: window size, window placement, input parameters, algorithm parameters, and view parameters. The Environment Control Panel is shown in Figure 12.

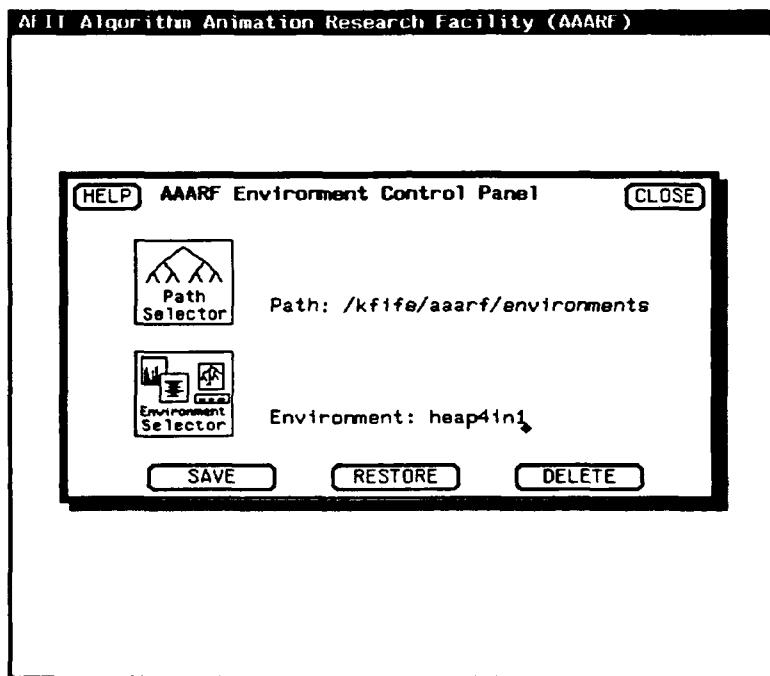


Figure 12: AAARF Environment Control Panel

Use the **Path Selector** to select a directory in which environments can be saved and restored. The path selector menu is activated by depressing the right mouse button on the Path Selector icon. The path can be changed only by the Path Selector; there is no option to enter the path from the keyboard. By default, the initial path is set to the user's current working directory. The path can be initialized to any valid directory by setting the the **AAARFENV** environment variable to the desired path. Typically users keep all their AAARF environments in a particular directory and set **AAARFENV** to that directory path. See [5] for more information on setting environment variables.

- *To save an AAARF animation environment, the user must have write privileges for the selected directory.*

Use the **Environment Selector** to select an existing environment name, or enter an environment name from the keyboard. Environment names consist of 1 to 15 alphanumeric characters. If a selected directory has no environment files, the **Environment Selector** menu does not pop up.

When the environment name is selected, click on **Save**, **Restore**, or **Delete** to perform the desired operation. Only 100 saved environments are permitted per directory, so it may become necessary to delete unused environments.

- *Save and delete operations are relatively fast, but restore operations can take several seconds depending on the current CPU and memory load.*

The environment control panel is not considered part of the environment with respect to save and restore operations. The panel can be closed only by clicking on the **Close** button.

4 Algorithm Windows

Animations take place in algorithm windows. When an algorithm window is first opened, a default data set, algorithm, and view are provided. These may be changed by opening the master control panel as described later in this section.

Up to four algorithm windows can be active simultaneously. Algorithm windows can be moved, sized, iconified, or destroyed at any time. Algorithm windows contain from one to four non-overlapping view windows for viewing the algorithm in execution.

- *Depending on the CPU load and the number of active processes, it may take several seconds for a new algorithm window to open. Be patient - give the workstation a few seconds to respond before trying again.*

4.1 Algorithm Window Menu

Press and hold the right mouse button with the mouse pointer anywhere on the algorithm window title bar to pop up the algorithm window menu. Figure 13 shows the Algorithm Window Menu. Move the cursor to select from the following choices:

- Iconify
- Master Control
- Animation Recorder
- Status Display
- Help
- Kill

Iconify

This selection causes the algorithm window to become an icon. While in the icon state, the animation is stopped. The algorithm window can be deiconified by clicking on the icon with the left mouse button or selecting Deiconify from the algorithm window menu.

Master Control

This selection causes the Master Control Panel to appear. The Master Control Panel provides the user a means for controlling the input, view, algorithm, and control parameters for the animation. Section 4.2 discusses the Master Control Panel in detail.

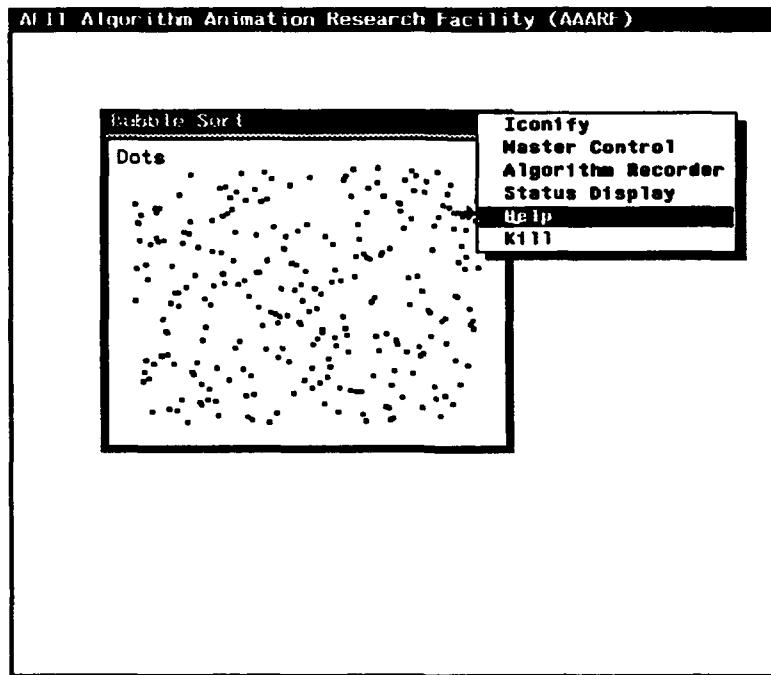


Figure 13: Algorithm Window Menu

Animation Recorder

This selection causes the Animation Recorder to appear. The animation recorder provides a means for creating, saving, and playing animation recordings. Section 4.3 discusses the Animation Recorder.

Status Display

This selection activates the algorithm status display which presents information regarding the algorithm's current state. The middle mouse button can be used to extract more detailed information regarding a particular element or area of interest within the algorithm. The Status Display is discussed in Section 4.4.

Help

This selection displays a help message which describes each of the options available from the algorithm window menu.

Kill

This selection permanently closes an algorithm window.

4.2 Master Control Panel

Figure 14 shows a typical Master Control Panel. The panel is divided into four separate but interdependent sections: the control section, the input section, the algorithm section, and the view section.

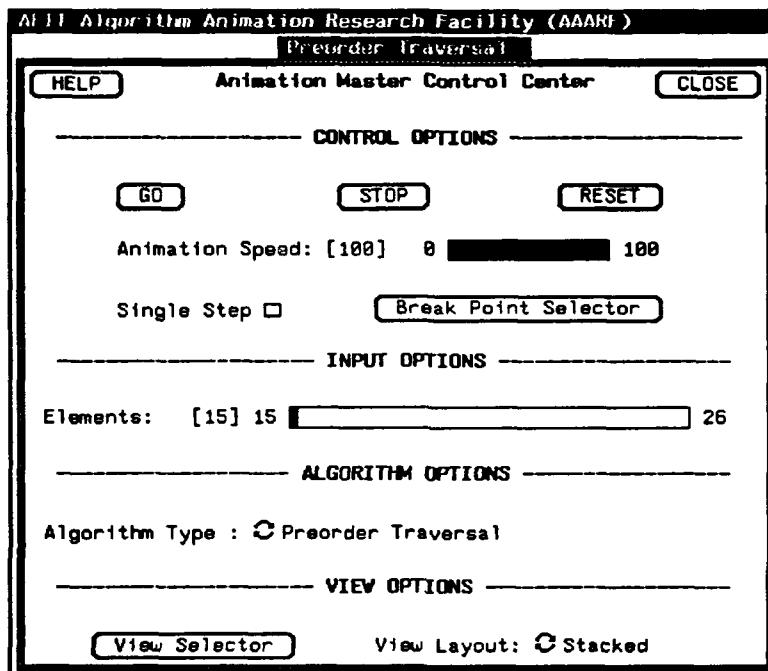


Figure 14: Master Control Panel for Traversal Algorithm Class

Control Section

The control section provides panel items for controlling the execution of the algorithm animation. The **Go**, **Stop**, and **Reset** buttons have obvious effects on the animation. To restart an animation which has run to completion, the **Reset** button must be hit. An alternative to these buttons is clicking the left mouse button in any of the open view windows. An algorithm which has run to completion can be reset by clicking the left mouse button in a view window.

The **Single Step** check box, the **Speed** panel slider, and the **Break Point Selector** are three more control mechanisms provided by the control section. Activating single-step mode causes the animation to stop after *every* IE. Selecting one or more break points causes the animation to stop after every IE which matches a selected break point. The speed control affects the execution speed of the animation; 100 is full speed, and 0 is very slow speed.

Input Section

The input section provides parameter controls to produce a variety of input data sets for a particular algorithm class. Changes to input parameters do not take effect until the animation is **Reset**.

Algorithm Section

This section provides, as a minimum, a panel cycle item for selecting the algorithm to be animated. There may be optional parameters for particular algorithms. For instance, quick sort options might be minimum partition size and the method for selecting a pivot value. Changes to algorithm parameters do not take effect until the animation is **Reset**.

View Section

This section provides, as a minimum, a panel selector for selecting from one to four views of the animation, and a panel cycle item for selecting the layout of the views within the algorithm window. There may be optional parameters for controlling the appearance of the view, such as the representation or size of nodes in a binary tree. Changes to view parameters take effect immediately.

4.3 Animation Recorder

Each algorithm window supports an animation recorder for recording and playing back computationally intensive animations. Figure 15 shows the AAARF animation recorder. The recorder also provides a convenient means for saving a particular set of algorithm parameters. Currently the recorder does not support playback in reverse.

The recorder is always in one of three states: OFF, RECORD, or PLAY. The algorithm window title bar reflects the current state of the recorder. Depending on the current recorder state, some operations may not be possible or allowed. Warnings are issued before any recording is erased.

The **Help** button activates a brief help screen explaining how to use the animation recorder. The recorder can be closed only by clicking on the **Close** button.

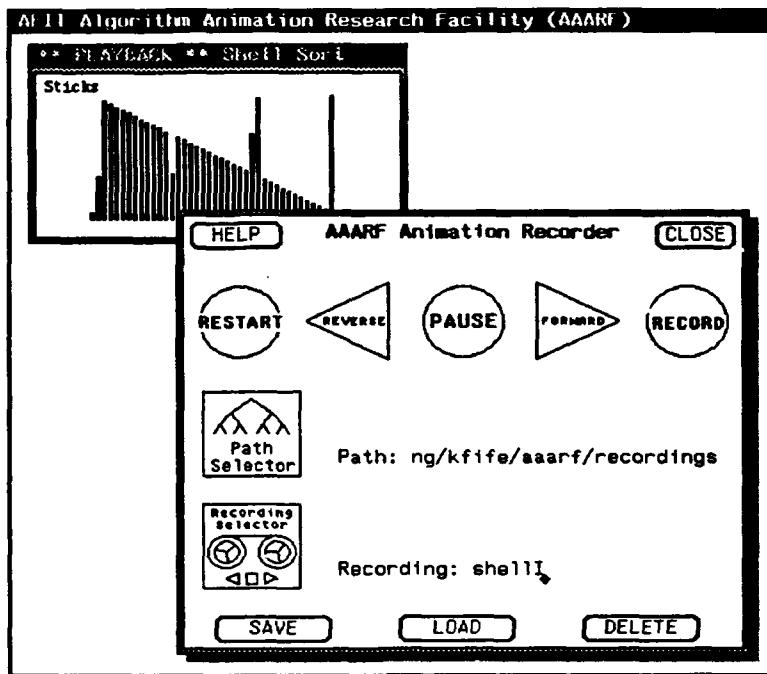


Figure 15: AAARF Animation Recorder

Recording

To record an animation, click on the RECORD button. The animation resets using the currently selected parameters, and the algorithm window title bar displays "**** RECORDING ****". Control the animation as usual. When the animation is finished, the recorder switches to PLAY mode and the algorithm window title bar displays "**** PLAYBACK ****". The recording can now be saved or played back.

- *Control commands are not saved; only the interesting events are recorded.*

Playback

Before a recording can be played, it must be recorded. Recordings can be saved, reloaded, and played; or just recorded and played without saving. Once a recording is loaded into the recorder, the algorithm window title bar displays "**** PLAYBACK ****". Playback can be started in several ways: the FORWARD button on the recorder, the GO button on the Master Control Panel, the GO button on the Central Control Panel, or by clicking the left mouse button in a view window. Since only the interesting events are recorded, the view parameters can be changed while the recording is playing back; likewise the recording can be controlled just like a "normal" animation.

File Functions

Use the **Path Selector** to select a directory in which recordings can be saved and restored. The path selector menu is activated by depressing the right mouse button on the Path Selector icon. The path can be changed only by the Path Selector; there is no option to enter the path from the keyboard. By default, the initial path is set to the user's current working directory. The path can be initialized to any valid directory by setting the the AAARFREC environment variable to the desired recording directory. Typically, users keep all their AAARF recordings in a particular directory, possibly partitioning it into subdirectories corresponding to each algorithm class. The AAARFREC environment variable is set to the parent directory name. See [5] for more information on setting environment variables.

- *To save an AAARF animation recording, the user must have write privileges for the selected directory.*

Use the **Recording Selector** to select an existing recording name, or enter an recording name from the keyboard. Recording names consist of 1 to 15 alphanumeric characters. If a selected directory has no recording files, the **Recording Selector** menu does not pop up.

When the recording name is selected, click on **Save**, **Restore**, or **Delete** to perform the desired operation. Only 100 saved recordings are permitted per directory, so it may become necessary to delete unused recordings.

Loading a recording sets the input, algorithm, and control parameters to those associated with the recording; the view parameters are not affected. Saving a recording resets all parameters to their state at the beginning of the recording.

4.4 Status Display

Each algorithm window supports a status display which provides information relating to the current state of the algorithm. Figure 16 shows a typical Status Display. The information and format may be different for each algorithm class. Users can interrogate the animation regarding specific aspects of the algorithm, such as a tree node or array element state by clicking the middle mouse button in a view window on a particular element of interest.

The Help and Close buttons perform their usual functions.

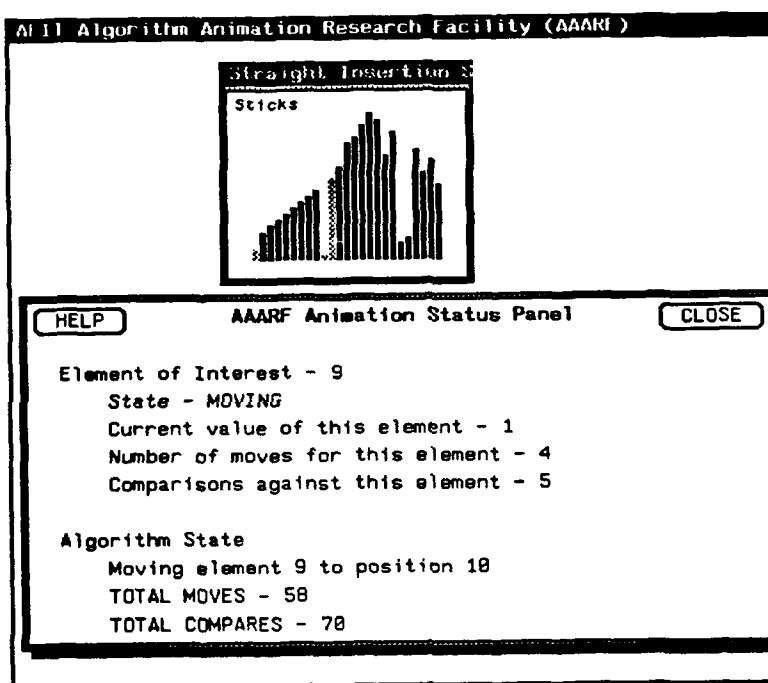


Figure 16: Status Display for ArraySort Algorithm Class

5 Parallel Algorithm

Using one of the parallel algorithm classes is similar to using a sequential animation. The difference is the added care that must be taken due to the link to the remote system that is hosting the algorithm. The AAARF processes must be able to communicate with the remote system to start the algorithm, as well as receive the data from it.

5.1 Remote System Set-up

The person running AAARF must have an account on the remote system. Furthermore, the account on the remote system must be set up to allow commands to be submitted from the workstation using `rsh`. This requires the presence of a file in the user's home directory called `.rhosts`. For the Intel iPSC/2 with UNIX System V Release 3.2, this file simply needs to contain a list, one per line, of the hosts that are allowed to remotely access the hypercube.

To make it possible for AAARF to run the algorithm remotely, two changes need to be made to the startup shell script; `.cshrc` for the `csh(1)` shell, and `.profile` for `sh(1)`. The first change entails adding the directory containing the executables for the `aaarf_clct`, `av`, and `server` programs to the `PATH` environment variable. The other change is to create a new environment variable called `AAARF_SYSTEM`. As the name implies, this must be set to the name of the Sun workstation that is used to run AAARF.

5.2 Running the Animation

Once the remote system is set up, the animation can proceed. If the animation is fully implemented, the rest of the session will proceed just as in Section 4. The Master Control Panel controls the algorithm on the remote system just as if it were on the workstation.

- *Due to the link with the remote system, the response time for starting a new run is quite long, sometimes as long as a minute or more — be patient!*

Some algorithms cannot be started remotely; sometimes the program is interactive, which makes it difficult for AAARF to control it. In these cases, the program must be started manually by someone logged into the remote system. The only difference this makes is that the program must also be terminated manually, since AAARF has no control over it.

Currently, it is not possible for AAARF to know whether the remote program started correctly or not. Some programs display messages indicating activity; these

messages are displayed in the window on the Sun workstation that is used to invoke AAARF, but the default configuration of AAARF uses the entire screen — preventing the user from seeing these messages. If these messages are useful, AAARF can be started with parameters that control the size of the area of the screen used by AAARF: the format is *aaarf width height*. The minimum size is 500 pixels square, about one-sixth of the screen in the upper left corner. By preventing AAARF from taking over the entire screen, the command window used to invoke AAARF is still visible, and messages from the remote program can be used to determine its status. It is also advisable to have another window active that is logged into the remote system so that the status of the program can be checked manually.

- *Do not attempt to start the animation by clicking the left mouse button in the window or by clicking the GO button on the Master Control Panel if the remote program has not started properly — AAARF will freeze and you must kill all programs on both systems manually.*

5.3 Parallel Performance Views

This section describes the performance views that are available in every parallel algorithm animation that uses the *parallel views library*[8:4-3]. Each of the performance views is described in the following paragraphs. The views are controlled by the **View Options** button in the **Performance View Parameters** section of the Master Control Panel.

The performance views contained in the *parallel views library* show basic information regarding message passing between processes, overall communications load, communications statistics for each node, and CPU utilization. These views are patterned after those in ParaGraph[4]; this is by no means an exhaustive set of performance displays, but merely a collection of displays that has proven useful. The views show different types of statistics, as well as different levels of detail.

Utilization This view is a histogram that displays the number of active processes on the vertical axis, and time on the horizontal axis. Figure 17 shows the utilization view during a run. The top part is colored in red to indicate the number of processors that are idle, and the bottom is green, giving the number of active processors. The *active* vs. *idle* state of a processor is derived from the message traffic: if a processor is blocked waiting for a message, then it is *idle*. Otherwise the processor is *active*.

Gantt This is a traditional view of processor activity. Each processor has a section of the vertical axis, and time is shown on the horizontal axis. When a processor is *idle*, that segment of the vertical axis is colored red; when the processor is *active*, it is

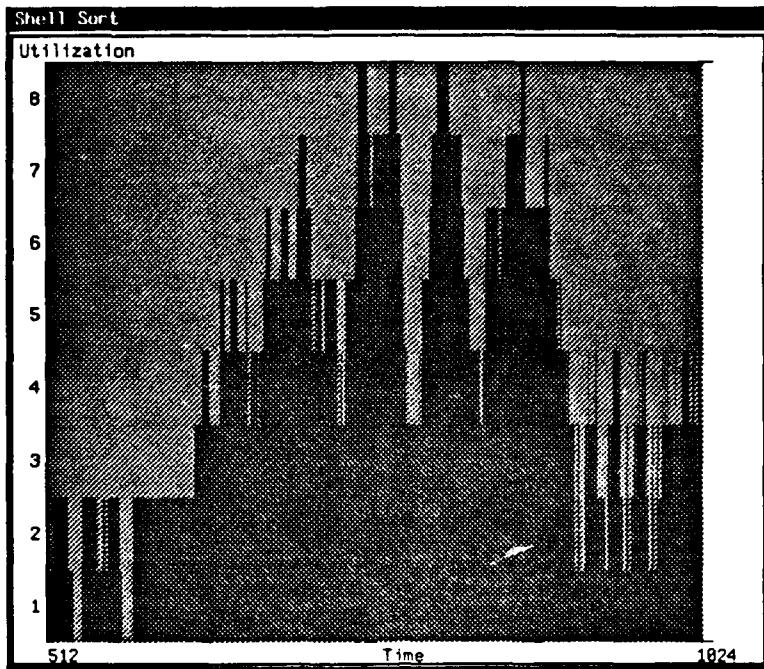


Figure 17: Sample Utilization View

green. The status of a processor is determined in the same way as for the utilization view. A sample Gantt view is shown in Figure 18.

Feynman The name for this view was taken directly from the corresponding Para-Graph display, though it may not have much relevance to the actual data being displayed. This view shows two different types of data: processor status and message passing activity. Figure 19 shows an example of this view. The processor status is shown in a similar fashion to the Gantt view, except that the data is shown as a thin horizontal line that is broken when the processor is blocked. Once again, the horizontal axis represents time. The message activity is shown by drawing lines between the horizontal lines representing the sending and receiving processors: The ends of the line are positioned according to the send and receive times. This clearly shows message passing patterns, delays, and bottlenecks. These displays are generated using trace records from send and receive system calls.

Communications Statistics This view provides a detailed look at the communications activity of a single (selectable) node. With time along the horizontal axis, the display is split into two parts: the upper half of the display shows outgoing message statistics, and the lower half shows incoming message statistics. Figure 20 shows a

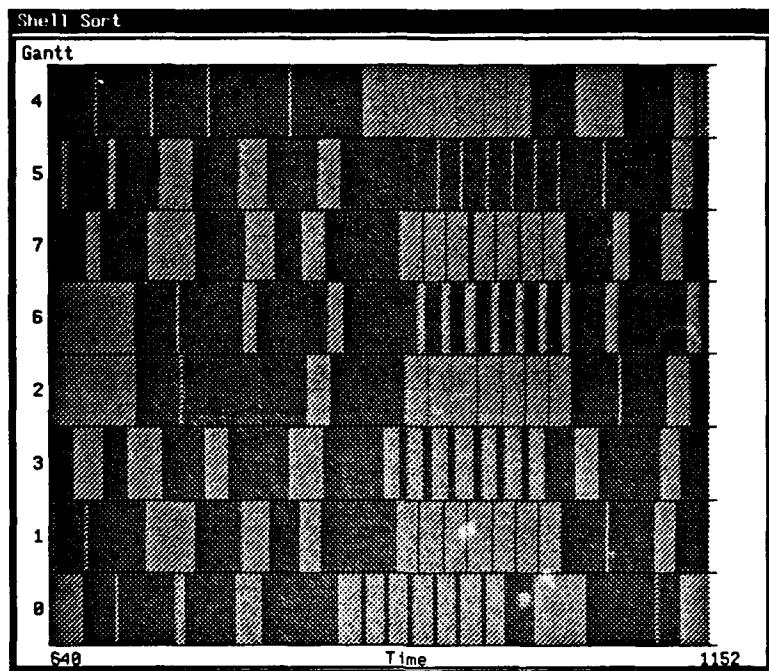


Figure 18: Sample Gantt View

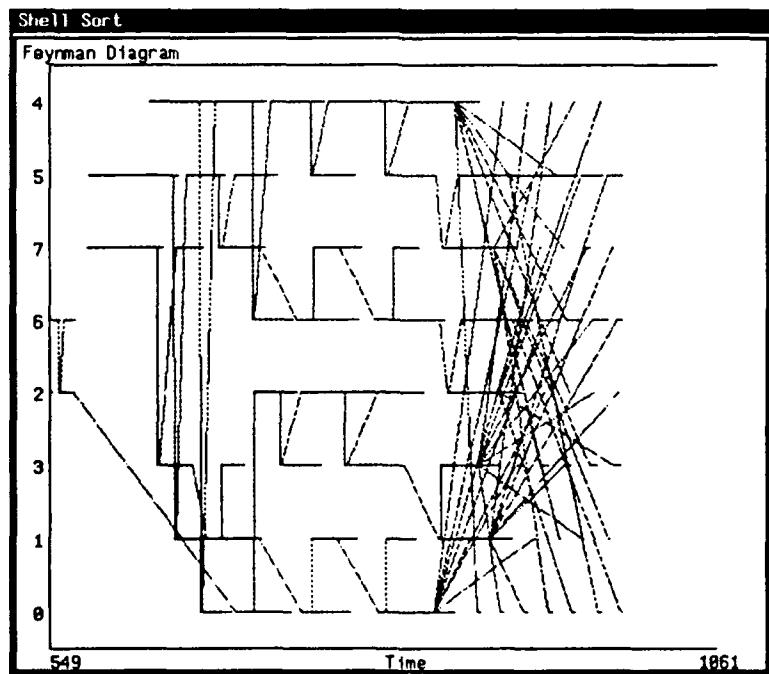


Figure 19: Sample Feynman View

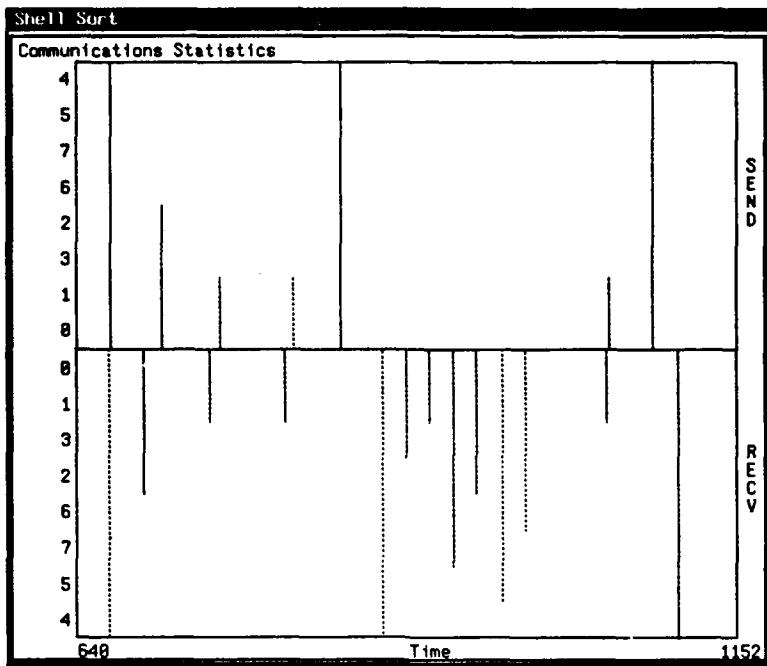


Figure 20: Sample Communications Statistics View

sample display. The data displayed can be chosen from three types: processor source or destination for each message, length of the messages, or the type of the messages.

Communications Load This view shows statistics on pending messages (messages that have been sent but not received) for the entire system versus time. Either the number of pending messages or the total length of all pending messages can be displayed. Figure 21 is an example of this view.

Queue Size Statistics for the input queue for one (selectable) node are displayed in this view. The presentation of the data is identical to the communications load view; The data is shown in relation to time, and either the number of messages or the total length of all messages in the input queue for the node is displayed. A sample of this view is shown in Figure 22.

Message Lengths This view uses a different method of presenting message passing information. The view contains a matrix, and a send operation causes an element of the matrix to be colored. The sending processor number determines the row and the receiving processor determines the column of the matrix. The length of the message determines the color used to fill in the element. See Figure 23 for an example.

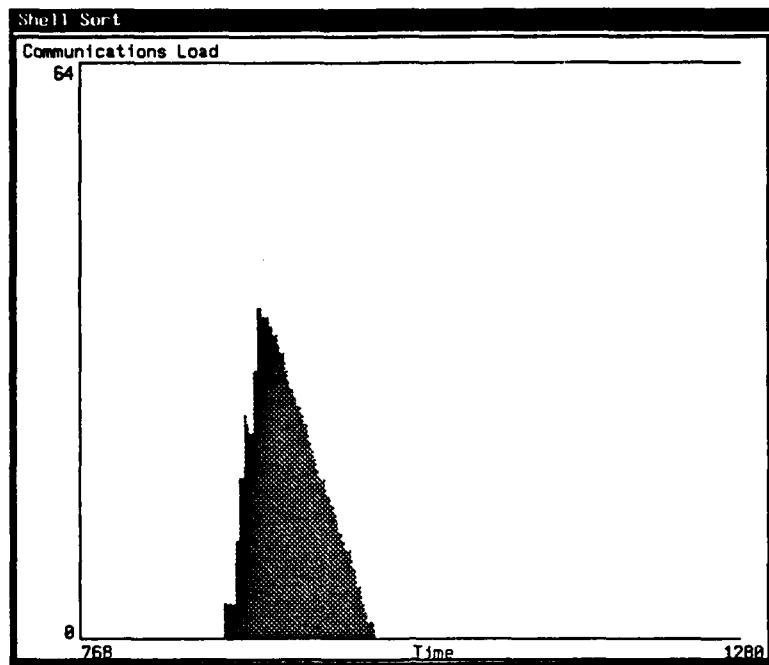


Figure 21: Sample Communications Load View

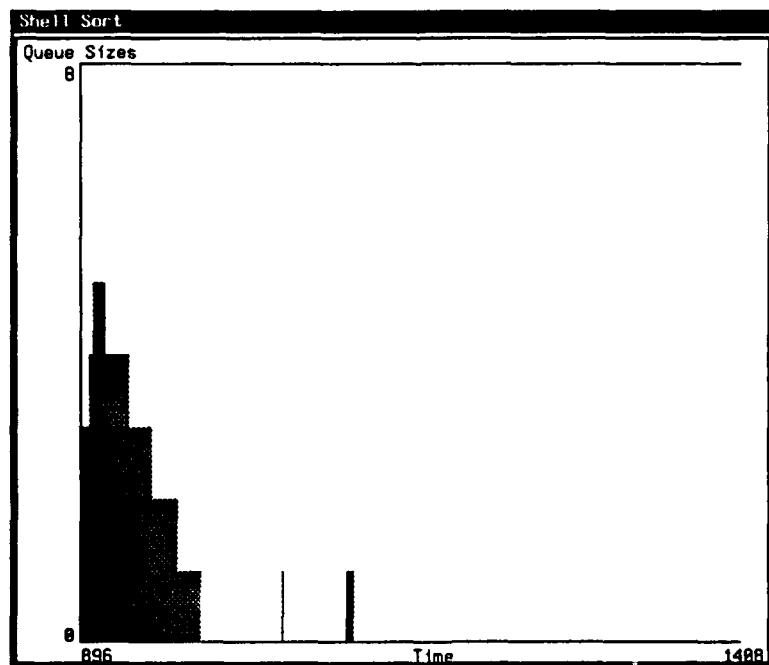


Figure 22: Sample Queue Size View

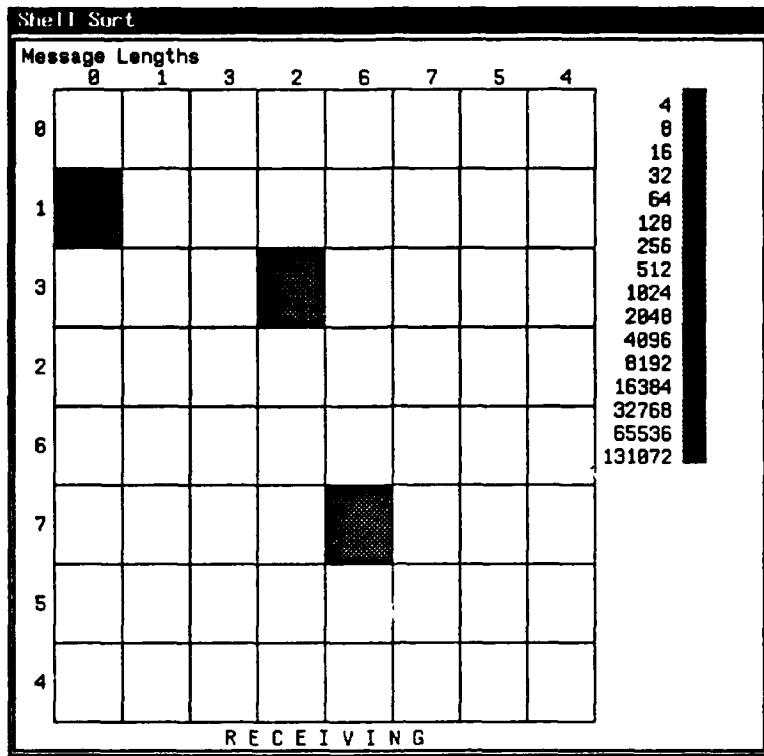


Figure 23: Sample Message Lengths View

Message Queues This view displays the current status of the input message queue for all the processors. As Figure 24 shows, the display is a histogram with the processors along the horizontal axis. The vertical axis can show either the number of messages in the queue or the total length of the messages in the queue. As the queue levels rise and fall, a 'high water mark' is left behind to mark the highest level that was attained for the run.

Kiviat This is very similar to a traditional kiviat chart — the processors are spread out around a circle and 'spokes' are drawn from each processor to the center of the circle (see Figure 25). The CPU utilization for each processor is calculated and plotted along the corresponding spoke of the wheel, with the center representing 0%. The polygon resulting from connecting points from adjacent spokes of the wheel creates a pattern that can be used to indicate the balance of the processing load among the processors. A 'high water mark' is also implemented here to indicate the highest level of processor activity.

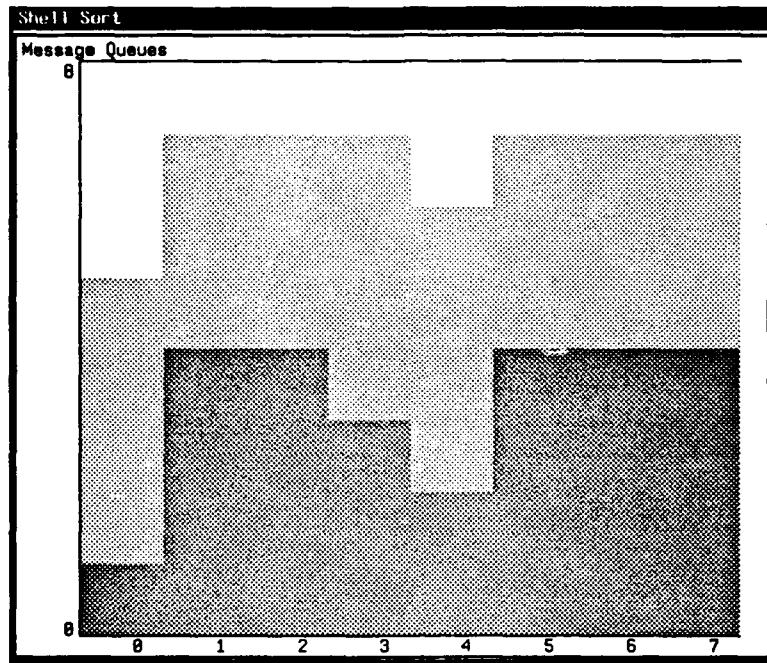


Figure 24: Sample Message Queues View

Animation Once again, the name for this view was taken from ParaGraph; the term, however, could be applied to any of the other views. This particular view shows the processors arrayed around the outside of a circle. The color of the processor indicates its status — idle, busy, blocked, or sending. In addition, lines are drawn to show message activity among the processors. When a message is sent, a line is drawn connecting the sending and receiving node; when the message is received, the line is erased. Figure 26 shows an example of this view.

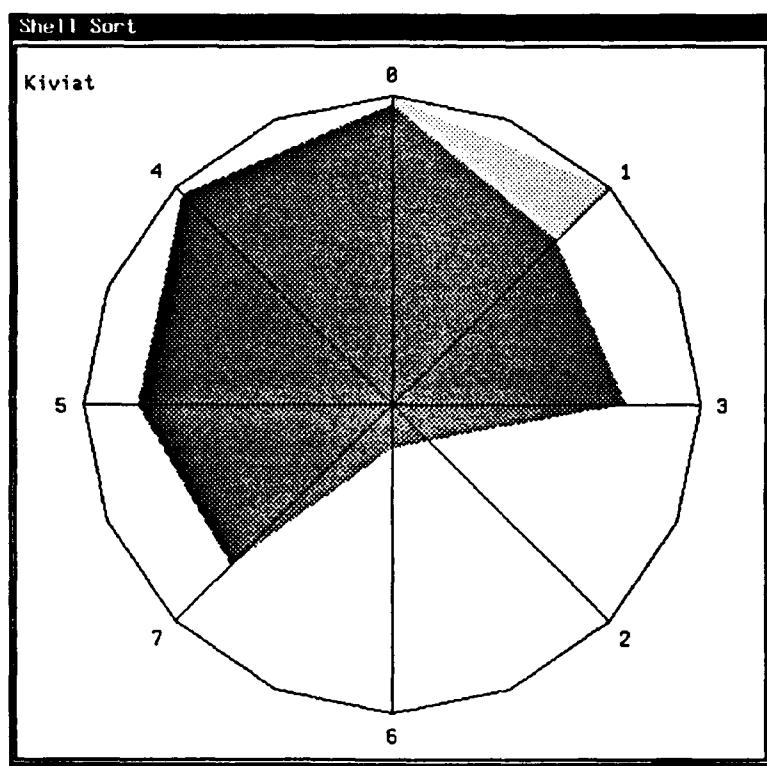


Figure 25: Sample Kiviat View

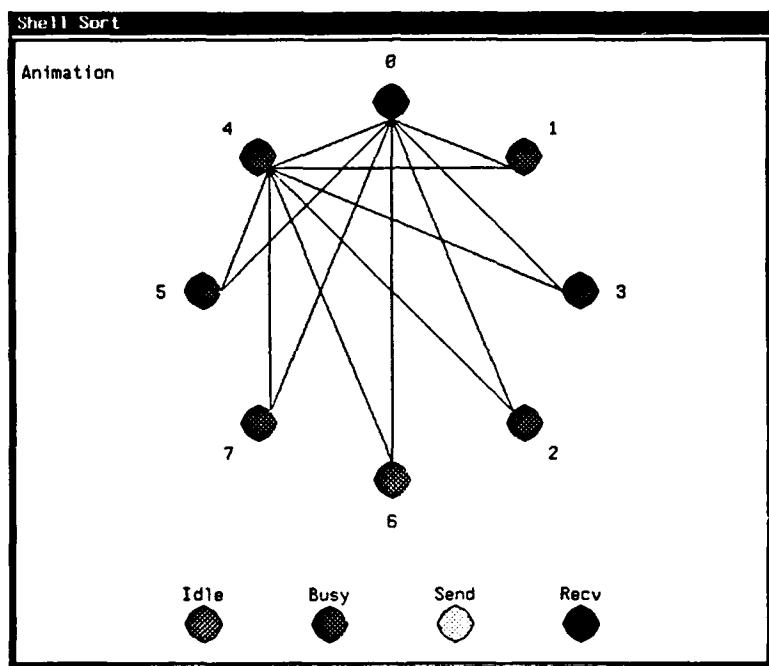


Figure 26: Sample Animation View

6 Exercises

This section presents exercises to test and develop your skill using AAARF. The first exercise directs you through an entire AAARF session. The remaining exercises require a little more thought.

Problem 1. Compare the performance of *Straight Selection Sort* to *Straight Insertion Sort* for sorting an array of 25 numbers. The numbers are already sorted, but in reverse order. Record the animation of the fastest algorithm. How many total moves were required? How many comparisons?

1. Startup AAARF by typing *aaarf* at the UNIX command line prompt.
2. Click on **CONTINUE** to close the welcome screen.
3. Open two ArraySort algorithm windows, by popping up the main menu, selecting the **New Algorithm Window** item, and then selecting **Array Sorts** from the pull-right algorithm menu.
4. Open the Master Control Panels for both windows.
5. Select *Straight Insertion Sort* for one window and *Straight Selection Sort* for the other.
6. Select 25 elements, 100% sorted, and Inverted order for both windows.
7. Select stacked Sticks, Moves, and Compares views for both windows.
8. Set speed to 100% in both windows.
9. Close both Master Control Panels.
10. Open the Central Control Panel.
11. Arrange the algorithm windows and Central Control Panel such that the algorithm windows are the same size, no windows are hidden, and both animations are easy to see and compare. Figure 27 shows a possible arrangement.
12. Open the Environment Control Panel. Select a directory and an environment name, and save this environment.
13. Close the Environment Control Panel.

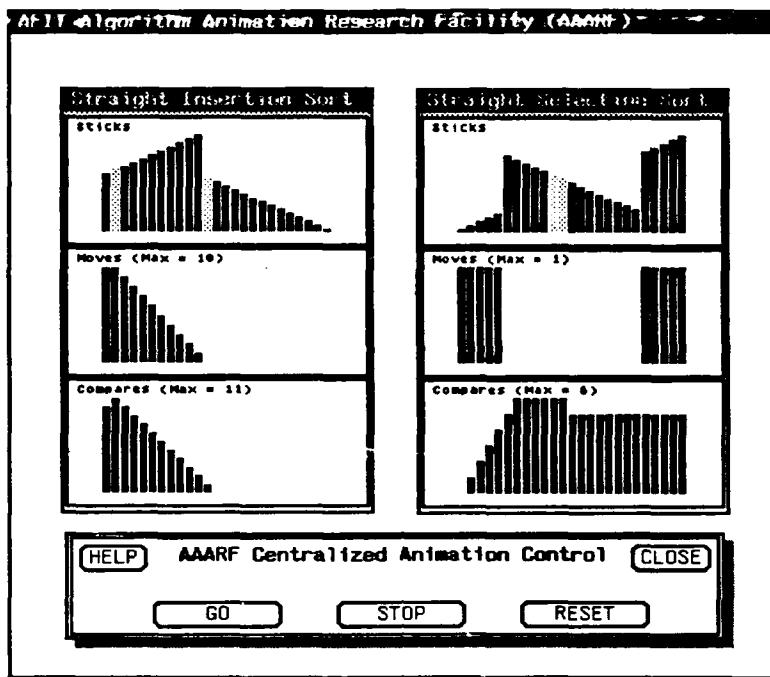


Figure 27: Possible Window Arrangement for Exercise 1.

14. Using the Central Control Panel, Reset both algorithm windows.
15. Using the Central Control Panel, start both algorithms by clicking on Go.
16. Determine which algorithm is fastest and close (Kill) the other window.
17. Open the Status Display for the fastest algorithm window and note the total comparisons and moves.
18. Close the status display.
19. Open the Animation Recorder.
20. Hit the Record button, then hit Forward.
21. When the animation is finished, save the recording.
22. Open the Master Control Panel and change the parameter settings. Hit the Reset button.
23. Load the recording that was just saved. Notice that the parameter settings reset to those associated with the recording.

24. Playback the recording.
25. Close the Recorder and Kill the algorithm window.
26. Pop up the AAARF main menu and Kill AAARF.

Problem 2. Open three *Traversal* algorithm windows. For each algorithm window, select 26 as the tree size, set the control to single step, and select both the tree and list views, but let the tree view dominate the algorithm window. Select a different algorithm for each window. Arrange and size the algorithm windows so that all three can easily be seen. Open the Central Control Panel and run the animations simultaneously. Save the environment. Which algorithm traverses the entire tree first? Which is last?

Problem 3. A program module to maintain a sorted array of numbers must be developed for the B-10 bomber automatic pilot software. The array of numbers is built by another module from five other sorted lists of numbers. Sometimes the five sub-lists are sorted in ascending order, other times in descending order. The sub-lists are always composed of at least 10 elements. Use AAARF to determine the best sort algorithm for this program module. Create an AAARF environment to compare the top two sort algorithms. Save the animation environment. Which algorithm is probably best?

Problem 4. Determine what effect the *reduction*, *lower bound*, and *dominance test* options have on the search process of the Parallel Set Covering Problem (SCP) algorithm. Choose a set of appropriate option combinations and make (at least) one run using one of the 20.20.x input data sets. Determine what factors should be used to compare the results of each run. Which option has the greatest impact?

References

- [1] Brown, Marc H. *Algorithm Animation*. Cambridge, Massachusetts: The MIT Press, 1987.
- [2] Fife, Keith C. "The AAARF Programmer's Guide." Air Force Institute of Technology, November 1989.
- [3] Fife, Keith C. *Graphical Representation of Algorithmic Processes*. MS thesis, Air Force Institute of Technology, 1989.
- [4] Heath, Michael T. "Visual Animation of Parallel Algorithms for Matrix Computations." In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.
- [5] Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.
- [6] Sun Microsystems. *Setting Up Your SunOS Environment: Beginner's Guide*, 1988. SunOS Technical Documentation.
- [7] Sun Microsystems. *SunView1 Beginner's Guide*, 1988. SunOS Technical Documentation.
- [8] Williams, Edward M. *Graphical Representation of Parallel Algorithmic Processes*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1990.

Appendix E. *SCP Source*

The source modules for the set covering problem implementation that were modified during the installation of the instrumentation are contained in Volume 2. The entire instrumented program, as well as the original SCP program can be obtained by contacting Dr. Gary Lamont at the following address:

Dr. Gary Lamont
AFIT/ENG
Wright-Patterson AFB, OH 45433

Phone: (513) 255-3450

email: lamont@afit.af.mil

Appendix F. *Source Code*

The source code for AAARF, the algorithm classes developed for this thesis, the instrumentation, and support programs is included in Volume 2 and is available through the AFIT library.

Bibliography

1. Aho, Alfred B., et al. *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.
2. AT&T. *UNIX System V/386 Release 3.2*, 1989. Generic System V/386 Manuals.
3. Beard, Ralph Andrew. *Determination of Algorithmic Parallelism in NP Complete Problems for Distributed Architectures*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, March 1990.
4. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes*, 11(4):11-14 (1986).
5. Brassard, Gilles and Paul Bratley. *Algorithmics: Theory and Practice*. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.
6. Brown, Marc H. *Algorithm Animation*. Cambridge, Massachusetts: The MIT Press, 1987.
7. Couch, Alva L. and others. *An Interactive System for Analysis of Hypercube Message Passing Performance*. Technical Report, Department of Computer Science, Tufts University, MA, 1986. Describes the SeeCube system for hypercube performance analysis.
8. Couch, Alva L. and David W. Krumme. "Monitoring Parallel Executions in Real Time." In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.
9. Cristofides, Nicos. *Graph Theory: An Algorithmic Approach*. London, England: Academic Press, 1975.
10. Fife, Keith C. "The AAARF Programmer's Guide." Air Force Institute of Technology, November 1989.
11. Fife, Keith C. "The AAARF User's Manual." Air Force Institute of Technology, November 1989.
12. Fife, Keith C. *Graphical Representation of Algorithmic Processes*. MS thesis, Air Force Institute of Technology, 1989.
13. Fox, Geoffrey C., et al. *Solving Problems on Concurrent Processors, Volume I*. Englewood Cliffs, New Jersey, 07632: Prentice-Hall, 1988.
14. Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, California: W. H. Freeman and Company, 1979.
15. Geist, G. A., et al. *PICL: A Portable Instrumented Communication Library*. Mathematical Sciences Section, Oak Ridge National Laboratory, May 1990.
16. Heath, Michael T. "Visual Animation of Parallel Algorithms for Matrix Computations." In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.

17. Horowitz, Ellis and Sartaj Sahni. *Fundamentals of Data Structures in Pascal*. Rockville, Maryland: Computer Science Press, 1987.
18. Intel Corporation. *iPSC/2 and iPSC/860 User's Guide*, June 1990.
19. Kahl, Mark Albert. *PRASE: Instrumentation Software for the Intel iPSC Hypercube*. MS thesis, Air Force Institute of Technology, 1988.
20. Kahl, Mark Albert. "The PRASE User's Manual." Air Force Institute of Technology, December 1988.
21. Kernighan, Brian and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Inc, 1978.
22. Kerola, Teemu and Herb Schwetman. "Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs.." In *Performance Evaluation Review - Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1987.
23. LeBlanc, Thomas J., et al. "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, pages 203-217 (September 1990).
24. Lee, Ann K. *Aggregate Logical Processes and Distributed Simulation Performance using the Chandy-Misra Algorithm*. MS thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1990.
25. Marinescu, Dan C., et al. "Models for Monitoring and Debugging Tools for Parallel and Distributed Software," *Journal of Parallel and Distributed Computing*, pages 171-184 (September 1990).
26. Myers, Brad A. *The State of the Art in Visual Programming and Program Visualization*. Technical Report CMU-CS-88-114, Carnegie Mellon University Technical Report, February 1988.
27. Paul, Richard F. and David A. Poplawski. "Visualizing the Performance of Parallel Matrix Algorithms." In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.
28. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.
29. Pountain, Dick. "The X Window System," *Byte*, 14:353-360 (January 1989).
30. Rover, Diane T. and Charles T. Wright. "Pictures of Performance: Highlighting Program Activity in Time and Space." In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.
31. Stasko, John T. "Simplifying Algorithm Animation Design with TANGO." Georgia Institute of Technology, June 1989.
32. Sun Microsystems. *Getting Started with SunOS: Beginner's Guide*, 1988. SunOS Technical Documentation.
33. Sun Microsystems. *Network Programming*, 1988. SunOS Technical Documentation.

Vita

Captain Edward M. Williams [REDACTED]

He graduated as Valedictorian from Mossyrock High School in Mossyrock, Washington in 1980, and obtained a Bachelor of Science in Electrical Engineering from the University of Washington, Seattle, Washington in December, 1984. Upon graduation, he received a commission in the US Air Force and was assigned to Lowry AFB, Colorado, where he served as a Software Engineer and Systems Analyst for a Satellite Sensor System. He entered the School of Engineering, Air Force Institute of Technology in May, 1989.

[REDACTED]

[REDACTED]

[REDACTED]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden estimate: 1 hour. This report is not subject to review and instructions, searching existing data sources, gathering and maintaining the data needed, and reporting information to the government. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing burden, to Department of Defense Headquarters Services, Directorate for Information Operations and Reports, 121 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to DOD Form 1 (GSA FPMR (41 CFR) 101-11, Appendix A and B) (OMB Control Number 1210-0188) Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED	
	December 1990	<i>Master's Thesis</i>	
4. TITLE AND SUBTITLE		5. FUNDING NUMBERS	
Graphical Representation of Parallel Algorithmic Processes			
6. AUTHOR(S)		7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	
Edward M. Williams, Capt, USAF		Air Force Institute of Technology, WPAFB OH 45433-6583	
8. PERFORMING ORGANIZATION REPORT NUMBER		9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	
AFIT/GCE/ENG/90D-07		Lt Col James Sweeder Strategic Defense Initiative Organization SDIO/ENA Room 1E149, The Pentagon Washington, DC 20301-7100	
10. SPONSORING / MONITORING AGENCY REPORT NUMBER		11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT		12b. DISTRIBUTION CODE	
Approved for Public Release; Distribution Unlimited.			
13. ABSTRACT (Maximum 200 words)		<p>Algorithm animation is a visualization method used to enhance understanding of the functioning of an algorithm. Visualization is used for many purposes, including education, algorithm research, performance analysis, and program debugging. This research applies algorithm animation techniques to programs developed for the Intel iPSC/2 hypercube. The goals for this visualization system are: Data should be displayed as it is generated. The interface to the target program should be transparent, allowing the animation of existing programs. The system should be able to animate any algorithm. The resulting system incorporates and extends two AFIT products: the AFIT Algorithm Animation Research Facility (AAARF) and the Parallel Resource Analysis Software Environment (PRASE). Since performance data is an essential part of analyzing any parallel program, views of performance data are provided as part of the system. Custom software is designed to interface these systems and to display the program data.</p> <p>While both <i>P-time</i> and <i>NP-time</i> algorithms can benefit from using visualization techniques, the set of NP-complete problems provides fertile ground for developing parallel applications.</p> <p>The program chosen as the example for this study is a member of the NP-complete problem set; it is a parallel implementation of a general Set Covering Problem (SCP).</p>	
14. SUBJECT TERMS		15. NUMBER OF PAGES	
Algorithm Animation, Parallel Performance, Program Visualization		246	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL